

C++ 17 从入门到精通

董洪伟 编著



清华大学出版社

C++17 从入门到精通

董洪伟 编著

清华大学出版社
北 京

内 容 简 介

本书以简明扼要的语言、配合丰富的实例，针对初学者从最基础的变量、表达式、数组、指针、引用和函数等，到面向对象的类和对象、继承与派生、虚函数与多态，从泛型编程的函数模板和类模板到移动语义、头等函数（函数指针、函数对象、Lambda 表达式），从 C++标准库的输入输出流库、容器、迭代器、算法、智能指针等工具到异常处理和 RAII 等，由浅入深地对最新的 C++17 标准语法进行了系统的讲解。对一些关键的语法概念如函数、类与对象、派生类等内容，提供了游戏编程、信息管理、数据结构、机器学习、人工智能等学科领域的一些经典的、实际问题的实战演练，以加强读者将语法知识用于解决各种实际问题和进行实际编程能力的训练，让读者领悟和体会 C++语言的灵活运用。

本书描述精炼、简单易懂，并有丰富的实战案例，既适合作为编程初学者的学习用书，也适合有编程基础的开发人员迅速学习和掌握现代 C++语言。

图书在版编目（CIP）数据

C++17 从入门到精通/董洪伟编著. —北京：清华大学出版社，2019
ISBN 978-7-302-52743-5

Ⅰ. ①C…　Ⅱ. ①董…　Ⅲ. ①C++语言—程序设计　Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2019）第 067169 号

责任编辑：闫红梅　张爱华
封面设计：刘　键
责任校对：徐俊伟
责任印制：宋　林

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：30.5　　**字　　数**：765 千字
版　　次：2019 年 8 月第 1 版　　**印　　次**：2019 年 8 月第1 次印刷
印　　数：1～1500
定　　价：79.00 元

产品编号：081355-01



前言

C++编程语言以其具有“可操纵底层硬件”“程序效率高”和“面向对象”的优势被广泛应用于系统软件和应用软件的开发，不但是企业界开发重量级软件或平台的首选语言，也是国内外高校广泛采用的计算机编程教学语言，更是衡量一个程序员功力的标尺。

然而，尽管企业界早已使用 C++11/14/17 标准，但国内高校仍然延用的是传统的、过时的 C++98 标准，已经和业界普遍使用的现代 C++语法标准有很大的脱节。

目前市场上还未见到国内作者编写的现代 C++语言教材，虽然有少量国外作者编写的现代 C++语言教材，但国外作者的思维模式和语言文化差异使得这些书难以被国内读者，特别是初学者阅读理解，这些书往往都是大部头的著作，令人望而生畏。由于 C++语言本身语法的复杂，琐碎的语法讲解使初学者感到枯燥乏味；缺少实践性的例子，初学者很难理解这些语法知识的价值和适用场景，不知如何将这些语法知识应用于实际编程中。这就造成了许多学过 C++语言的计算机专业的本科毕业生实际并没有掌握最基本的 C++编程知识，缺乏实际应用编程的能力。

本书作者在 C++课程的教学中深感缺少一本适合中国读者的、没有冗余语句、注重实战的现代 C++教材。于是参考了各种英文教材和网上资料，编写了这本面向初学者的、遵循 C++17 标准的语法与实践结合的入门教材，并且书中的实战案例对于有经验的程序员也很有参考价值。

本书的编写遵循下面几个目标。

（1）针对没有任何编程基础的学生，直接讲解最新的 C++17 标准，避免传统的从 C 到 C++的教学模式和国内高校采用的过时的 C++98 标准语法，使读者可以直接学会使用最新的现代 C++语法特征，如 auto、range for、Lambda、移动语义、变长模板等，避免了 C 和 C++比较式教学带来的混乱，也不需要浪费时间在 C++旧标准语法上，可使无编程基础的初学者在较短时间内快速掌握现代 C++编程语言的核心内容。

（2）突出重点，讲解主要的常用语法，而不是一本面面俱到的语法手册，由浅入深、由易到难，尽量用浅显易懂的例子说明语法概念，力求简明扼要，避免空洞的概念和冗长的描述。

（3）从入门到实战，只有通过具体、长期的实战训练，才能逐步熟练精通一门编程语言。

语法知识可能短时间就能理解，但只有通过大量的实战训练才能真正熟练使用一门编程语言。本书准备了从游戏编程、信息管理、数据结构、机器学习、人工智能等不同领域的一些经典的实战案例，希望这些案例能够帮助读者消化语法知识、提高学习兴趣，逐步将 C++用于解决各种实际问题，避免出现“只会考试而不会编程”的普遍问题。

由于实战案例涉及一些其他学科专业知识，初学者和教师可以根据自己的需要选读实战部分，甚至完全跳过实战案例也不影响 C++17 语言的教学。本书的源代码可以登录清华大学出版社网站(www.tup.com.cn)下载。

由于作者知识和水平所限，错误之处在所难免，欢迎读者批评指正。

作者

2019 年 1 月


目录



C++介绍

1.1 程序与编程语言

1.1.1 计算机是什么

计算机是一种根据指令对数据处理的通用计算设备。每台计算机都有一个称为中央处理器(CPU)的微处理器芯片执行对数据处理的指令,不同计算机的指令集是不一样的。

1. 计算机指令

计算机接受一系列指令作为输入,逐个处理它们,并且通常显示某种输出表示它已完成的操作。这类似于人们日常生活中通过一系列操作步骤完成一个任务的方式,如一个人通过如下一系列步骤完成“做饭”的任务。

(1) 从容器(米桶)中取出米,放入洗米盆中。

(2) 用自来水对洗米盆中的米进行冲洗。

(3) 如果电饭锅没洗净,洗净电饭锅。

(4) 打开电饭锅盖,将米和水放入电饭锅中。

(5) 插上电源,按下开关。

(6) 饭好后,拔下电源(任务结束)。

虽然人们可以理解自然语言(如英语)中的复杂指令,但计算机只能理解为用计算机语言表达的非常简单的机器指令集中的指令。不管多么复杂的计算,在计算机内部都是被分解成许多简单的、逐条执行的机器指令。告诉计算机如何执行复杂任务的指令序列称为**程序**。

以下是一些简单的计算机指令示例。

(1) 算术:加、减、乘、除法。这些通常被称为**算术操作**。

(2) 比较:比较两个数字,看哪个较大或者它们是否相等。这些通常被称为**逻辑操作**。

(3) 分支:跳转到程序中的另一条指令,并从那里继续。这些通常被称为**控制语句**。

2. 计算机的组成部分

1）计算机的主要类型的组件

计算机包含4种主要类型的组件。

(1) 输入：允许计算机从用户接收信息的任何设备，包括键盘、鼠标、扫描仪和话筒等。

(2) 处理：处理信息的计算机组件。计算机的主要处理部件是中央处理器(CPU)，但在现代计算机中也可能有其他处理器。例如，许多图形卡都带有图形处理器(GPU)，GPU以前只用于处理图形，现在也可用于通用程序。

(3) 存储：存储信息的组件，包括主存储器(也称为内存)和二级存储器(如硬盘驱动器、CD或闪存盘等外部存储器)。存储器是存储运行程序的指令和数据的地方。

(4) 输出：用于向用户显示信息的任何设备，包括显示器、扬声器和打印机。

2）举例理解计算机

可以通过自动售票机来理解计算机(尽管自动售票机严格地说并不是计算机)。

(1) 输入：投币口和选择按钮是自动售票机的输入设备。

(2) 处理：当进行选择后，自动售票机会执行以下几个步骤——验证是否有满足条件的票、验证身份信息、检查和验证是否收到足够的资金、修改数据库、计算差额。执行所有这些步骤的机器都可以被认为是处理器。

(3) 存储：自动售票机需要在某个地方保存信息，如票的库存、价格等。

(4) 输出：自动售票机显示结果、打印票。

3. 中央处理器

中央处理器(CPU)是计算机中最重要的部分，是计算机的大脑，负责计算、处理数据、控制其他设备等。它有几个重要的子组件，具体如下所示。

(1) 算术逻辑单元(ALU)：执行算术和比较操作。

(2) 控制单元：确定下一个要执行的指令。

(3) 寄存器：形成一个高速存储区以保存临时结果。

不同种类的CPU可以理解不同的指令集。例如，Intel IA-32、x86-64、IBM PowerPC和ARM。

4. 存储器

计算机将信息(程序、数据)存储在存储器(memory)中，存储器有两种类型：**主存储器**(也称为**内存**)和**辅助存储器**(也称为**外存**)。

主存储器直接连接CPU(或其他处理单元)，通常称为**RAM**(**随机存取存储器**)。计算机关闭时，大多数主存储器都会丢失其内容，即具有易失性。

可以将主存储器想象成一个排成一列的存储器单元，每个单元都可以通过其存储器地址寻址。对于第一个单元，地址从零开始，并且每个后续单元的地址比它之前的地址多一个，正如一个班级中学生的学号从1开始依次递增。每个单元只能保存长度固定的用二进制表示的数值，但CPU可以随时用新的数值替换原有内容。

辅助存储器比主存储器便宜，但可以存储更多的内容。虽然它慢得多，但它是非易失性的，也就是说，即使在计算机关闭后其内容也会保留，如硬盘和闪存盘。

计算机的操作系统提供操作辅助存储器的高级接口。这些接口允许信息以文件的形式

保存在辅助存储器中，并且将文件组织成目录的层次结构。接口和层次结构通常被称为**文件系统**。例如，Windows 系统使用资源管理器等工具访问这些目录和文件。

1.1.2 计算机编程

1. 算法

算法(algorithm)是完成某个任务或解决某个问题的一系列步骤(指令)，如一道菜的制作过程说明、祖冲之计算圆周率的方法等。

2. 程序和编程

程序就是算法在计算机中的表示和实现。编程就是如何用计算机的指令来表示算法，即将算法转换成计算机可以执行的程序。

3. 二进制

因为计算机硬件是由很多晶体管组成的，而晶体管只有“开”和“关”2 种状态，因此一个晶体管只能表示 2 个数字 0 和 1，通过很多个这种表示 0 或 1 的晶体管可以表示更复杂的数值，如整数或字符等。不管多么复杂的程序数据，在计算机硬件中都是以二进制(0 和 1)形式表示的。

1 个晶体管器件只能表示 1 位二进制数(0 或 1)，称为 1 **比特(b)**或 1 位。8 个器件可以表示 8 位二进制数字，即可表示 2^8 个不同的数值。8 位二进制数称为 1 **字节(B)**。16 个器件可以表示 16 位二进制数，即 2B，……

8×1024 个器件就可以表示 1024B，即 1KB。

8×1024×1024 个器件就可以表示 1024KB，即 1MB。

8×1024×1024×1024 个器件就可以表示 1024MB，即 1GB。

4. 机器语言

计算机中的指令和数据都是用 0、1 串表示的。机器语言(machine language)是用这种二进制代码表示的计算机能直接识别和执行的一种机器指令集合。

例如，下面是将 17 和 20 相加的机器指令(采用 Intel 8086 机器语言，Intel Pentium 机器语言的子集)。

```
1011 0000 0001 0001
0000 0100 0001 0100
1010 0010 0100 1000 0000 0000
```

第一行告诉计算机将 17 复制到 AL 寄存器：前 4 个字符(1011)告诉计算机将信息复制到寄存器中，接下来的 4 个字符(0000)告诉计算机使用名为 AL 的寄存器，最后一个 8 位二进制数(0001 0001，即表示整数 17)指定要复制的数。

用机器语言编写程序非常困难，也很难被人们阅读和理解。在 20 世纪 40 年代，程序员必须这样做，通过纸上打孔表示 0 和 1 来编写机器语言的程序，因为没有其他选择。

5. 汇编语言

为了简化编程过程，引入了汇编语言(assembly language)。每个汇编指令对应一个机器语言指令，但人们更容易理解，如上述的二进制指令用 8086 汇编语言的指令等效表示

如下。

```
MOV AL, 17D
ADD AL, 20D
MOV [SUM], AL
```

用汇编语言编写的程序不能直接被计算机理解，因此需要翻译步骤。汇编程序可以将汇编语言编写的程序转换为机器语言程序。

6. 高级语言

虽然汇编语言对机器语言有很大的改进，但它仍然很神秘，而且它的级别太低，和机器语言一样，执行一个最简单的任务也需要很多指令。于是，人们发明了高级语言（high-level language），使编程变得更加容易。

在高级语言中，一个指令可以对应多个机器语言指令，高级语言采用类似人类的语言表达指令，使得程序更容易理解和编写。下面是用C++语言编写的与上面机器（汇编）代码等效的代码。

```
sum = 17 + 20
```

1.1.3 编译器、解释器和C++语言

用高级语言编写的程序在计算机执行之前必须翻译成机器语言。一些编程语言的程序被整体翻译成机器语言的程序并存储在另一个文件中，然后执行，这种语言称为**编译型语言**。也有些语言的程序语句被逐条翻译，逐行执行，这种语言称为**解释型语言**。C++是一种编译型语言。

编译型语言通过一个**编译器**程序，将该语言编写的源文件编译为可执行的二进制程序文件。解释型语言通过一个**解释器**程序，对该语言编写的源文件的每一句都逐条解释并执行。

- 编译器：是将整个源代码程序一次性全部转换为机器指令代码的工具。转换后的机器语言代码可以直接在计算机上运行。
- 解释器：是一行一行地、逐条地将源程序语句转换成机器指令并执行。从第一条语句开始，转换一条语句后就执行，然后再转换并执行下一条语句，……

解释器逐条语句转换并执行，使初学者很容易知道程序的错误位置。编译器对整个源程序进行一次性转换，其优点就是可以对代码进行整体的优化，从而提高程序的性能。

C++语言是对C语言的面向对象扩展，对运行速度要求高或需要直接操纵硬件的程序通常使用C语言或C++语言编写。编译器将C++语言程序编译为机器指令时，会对程序做很多细粒度的控制和优化，可以提高程序的运行速度。

1.1.4 C++语言介绍

C++语言是由Bjarne Stroustrup于1979—1980年在贝尔实验室发明的编程语言。C++语言对古老的过程式编程语言——C语言进行了改进、扩展，增加了完善的面向对象语言特

征。C++可运行于多种平台上，如 Windows、Mac OS 以及各种版本的 UNIX（包括 Linux）操作系统上。

作为最常用的几种著名编程语言之一，多年来，C++语言一直稳居各种语言排行榜的前 5 名，C++可直接操纵底层硬件，C++程序和 C 程序具有同样高效率的性能和优点，不但可以用于与硬件相关的系统程序的开发，也用于对速度性能要求高的各种应用程序和平台软件的开发。这些系统和应用程序包括各种操作系统，设备驱动程序，嵌入式程序，游戏引擎及大型游戏，高性能科学计算（如气象预测、地理信息系统、人工智能计算平台），计算机辅助设计与制造（CAD/CAM），图形图像和动漫软件，操作系统的应用软件（如浏览器、Office），银行金融证券系统，Web 服务器等。

C++的面向对象语言特征支持现代的面向对象设计和编程思想，并且还提供了功能强大的 C++标准库，其中以模板形式实现的通用的数据结构容器和算法，极大地提高了程序的开发效率。越来越多的 C++程序员已经抛弃过时的 C 风格编程，改为使用先进的 C++。

自 20 世纪 80 年代，C++语言产生并标准化后，尽管 C++语言一直是活跃的主流编程语言，但是语言自身的改进相对缓慢，直到 2011 年，ISO 发布了新的 C++11 标准，标志着现代 C++语言的诞生。C++11 对传统的 C++语言做了很多本质上的改进，可以说相当于一个新的编程语言，增加了许多优秀的新的语言特征，如 Lambda 表达式、auto 关键字、range for 等，标准库提供了更完善的容器、算法、智能指针等工具。

C++11 点燃了 C++社区的热情，此后每 3 年就产生一个新的标准，2014 年的 C++14，2017 年的 C++17，目前，C++20 标准已经制定。工业界在产品开发中已经普遍采用现代 C++，各大主流编译器都已经支持 C++17 的绝大多数语言特征。但大学的 C++课程已经落伍于工业界，仍然沿用传统的老式 C++语法。跳过 C 语言、传统 C++，直接学习 C++17 不但可以节省学习时间、提高学习效率，更符合工业界的需求。

1.1.5　C++程序开发步骤

1. 开发步骤

和其他任何编程语言编写程序一样，用 C++编写程序同样要经历以下步骤。

（1）理解问题：是一个什么样的问题？输入数据是什么？要产生什么结果？

（2）提出算法：解决这个问题的指令（步骤）序列。

（3）编写程序：将算法转换成某种编程语言的程序。

（4）测试：各种可能性的不同的输入，是否产生预期的结果。

2. 举例说明开发步骤

例如，要计算一组数值的平均值，可以按照以下步骤进行。

（1）理解问题：这些数值从哪里（键盘还是文件）输入？结果如何显示（屏幕打印输出还是保存到文件）？

（2）提出算法：用 2 个数值分别表示总和与数值的个数，然后将输入的这些数累加到总和上，最后除以数值的个数，得到平均值。人们经常以伪代码的方式描述算法的过程。

```
----------- start -------------
总和 sum = 0
```

```
计数器 count = 0
重复:
    读一个值
    如果读取值失败,结束这个"重复"过程
    否则:
        将读取的值 value 加到 sum.
        计数器增加 1.即
            count = count + 1
通过"总和"和"计数器"相除得到平均值
显示/打印平均值
------------ end ------------
```

(3) 编写程序:将算法用 C++语言表示出来。

(4) 测试:输入不同的测试数据,看看结果是否正确。输入的数据可以包含非法的数据,看看程序是否能适当地应对,如输入的是字符串而不是数值,程序是否会提示等。

1.2 C++程序结构

1.2.1 最简单的C++程序

```
int main() {
    return 0;
}
```

这是一个最简单的 C++程序,程序一执行就结束了。

1.2.2 函数

C++程序是由一些**函数**构成的,每个 C++程序都执行唯一的叫作 main()的**主函数**。这个函数里花括号{}包围的部分就是这个函数的程序语句。

上述的 main()函数中只有一条以分号;结尾的称为**语句的指令**"return 0;"。return 是返回的意思,该语句表示函数返回值 0,也就是说这个函数执行该语句返回一个结果 0 给这个函数的调用者,因为 main()函数是由操作系统调用的,因此,这个结果 0 就返回给操作系统。操作系统可以根据这个返回结果,判断程序是正常执行还是出现了错误。一般地,非 0 的整数表示某种错误代码,不同操作系统对返回的整数代码表示的错误有不同的约定。

main()函数名左边的 int 表示函数返回结果的数据类型,即结果 0 是一个 int(整数)类型的值。

main()函数名右边的圆括号()表示可以给这个函数传递一些参数(即输入数据)。

除了 main()函数,C++程序中还会包含其他不同名字的函数,每个函数都包含由花括号{}包围的一些程序**语句**。

一个函数的定义格式是:

```
返回类型　函数名(参数列表)
{
    (多个语句构成的)函数体
}
```

即一个函数包含 4 部分。

- 返回类型：说明这个函数执行后返回的结果值的数据类型。如 main()函数前的 int 表示这个程序的返回结果是 int 类型的值。
- 函数名：函数的名字，如 main。
- 参数列表：由“(”和“)”括起来的形参(以后会介绍)。
- 函数体：由“{”和“}”括起来的 0 条或多条程序语句。上述代码中的 main()函数体只有一条语句，一般函数的函数体中通常会有多条语句。

1.2.3　语句

C++程序的最小的完整执行指令都是以分号结尾的语句，如 main()函数的“return 0;”语句。可以将一条语句写在多个行，不管中间有多少空格、回车符、换行符，最后都是以分号作为语句的结束。因此，下面的程序和上面的程序是一样的。

```
int main() {
    return
        0;
}
```

1.2.4　程序注释

为了方便他人阅读程序或自己回顾程序，可以在程序中添加一些对程序进行说明的文字——**注释**。

注释本身不是代码，仅仅是对程序代码的说明。完全可以删除注释，而不影响程序的任何功能，但良好的程序应该添加一些注释以方便阅读理解。

注释主要分为多行注释和单行注释。

- **多行注释**(也称为**块注释**)以/*开头，以*/结尾。它们之间的一行或多行文字都是程序注释。
- **单行注释**用//开头，其后的同一行的文字都是注释。

如上述代码可以添加如下注释。

```
/*
  ------这是多行注释----------
  这是我的第一个 C++程序
  this is my first C++program
  作者：hwdong
  修改日期：2018-01-27
*/
```

```
int main() {            //main()函数是程序的入口,程序总是从这里执行
    return 0;           //程序结束,返回整数 0
                        //C++的程序语句都以分号';'结尾
}
```

注意：块注释不能嵌套,即不能在块注释中再出现/＊或＊/。下面的块注释是错误的。

```
/*
  块注释不能嵌套,不能在块注释中间包含/*或*/
  作者：hwdong
  修改日期：2018-01-27
*/
```

1.2.5 hello world 程序

```
#include <iostream>
using namespace std;
int main() {
    cout <<"hello world";
    return 0;
}
```

执行这个程序,会在控制台窗口输出字符串"hello world"。如何执行 C++程序,请看本书作者网站的文章或视频。

1.2.6 标准输入输出库和 cout

C++程序员在编写 C++程序时,不可能所有程序代码都自己从头编写,经常会调用别人已经编写好的代码,这些编写好的代码通常以“C++库”的形式存在,程序员在自己编写的程序中调用各种库提供的现成代码,可以极大地提高程序开发的效率和质量。这些库通常都经过了专门的优化和测试,具有很好的效率和可靠性,如果程序员什么都从头编写,不但效率低也容易出错。除了各种第三方库外,还有 C++标准委员会制定的 C++标准库。

C++的各个具体实现都自带了 C++标准库的实现,程序员不需要进行任何安装和配置就可以使用 C++标准库。这些库又按照功能进行了不同的划分,如其中的 C++标准输入输出流(库)就包含了针对标准输入输出设备(屏幕窗口、键盘)的输入输出的一些具体工具(函数、类、对象)。要使用其中的某个工具,就要包含对这个工具说明的**头文件**。例如要使用标准输入输出流中的各种工具,就需要在程序中**包含**(**include**)标准输入输出头文件 iostream (stream 是流的意思,io 是输入 input 输出 output 的缩写):

```
#include <iostream>
```

这是一个**预处理指令**而不是 C++指令语句。C++编译器在编译源代码前,会使用一个预处理工具对这种以#开头的预处理指令进行处理,**#include** 指令称为**包含预处理指令**。

预处理工具遇到这种指令时，会将 **#include** 后面文件名指示的文件的内容包含到程序中来，即用文件 iostream 的内容替换掉这个预处理指令。

一旦用**包含预处理指令**包含了 iostream 头文件，程序中就可以使用这个头文件中声明的各种对象，其中 cout 对象代表的是标准输出流对象(即代表终端窗口)。

C++的输入输出将输入输出看成是数据向输入输出设备的流动的过程。可以用输出运算符<<向 cout 代表的标准输出对象(窗口)输出信息，如输出一个字符串：

```
cout <<"hello world";
```

用双引号括起来的一系列字符在 C++中称为**字符串**。

1.2.7　名字空间

一个 C++程序可能使用他人写的代码或库，不同的程序库可能会用同一个名字表示某种对象、函数等，即会出现名字冲突。正如日常生活中遇到 2 个同名的叫作“张伟”的人。C++为了防止这种不同代码或库之间的名字冲突，引入了名字空间的概念，即 C++规定每个名字都属于某个名字空间。

如可以用“数学 1701 班”的“张伟”和“计算机 1805 班”的“张伟”来区分它们。“数学 1701 班”和“计算机 1805 班”就是 2 个不同的名字空间。

C++自带的标准库中的所有对象、函数等都属于一个叫作 std 的**标准名字空间**。如上面的 cout 变量(对象)就是属于标准名字空间 std 中的一个名字。通过在程序代码前使用“**using namespace std;**”将整个标准名字空间 std 的名字都引入到程序中，从而告知编译器这些名字的含义。如果缺少这一条语句，编译器会报错：“cout 未定义”，也就是说编译器不认识这个 cout。

也可以不使用“using namespace std;”引入 std 中的所有名字，而是在名字(如 **cout**)前加上名字空间名和 2 个冒号::构成的**名字空间限定(std::)**。例如：

```
#include <iostream>
int main() {
    std::cout <<"hello world";
    return 0;
}
```

std::cout 表示这是名字空间 std 的 cout。

某个名字如函数名 main 没有明确说明名字空间，那么它就属于一个**全局名字空间**。全局名字空间中的名字不需要名字限定。

再如 std::endl 是标准名字空间 std 中的某个名字 endl，它表示的是换行字符(简称换行符)，输出运算符<<对这个符号会执行一个换行的动作，即从新的一行开始新的输出。

```
#include <iostream>
int main() {
    std::cout <<"hello world";
    std::cout << std::endl;
```

```
    std::cout << "教小白精通编程";
    return 0;
}
```

执行该程序后，将在控制台窗口输出：

```
hello world
教小白精通编程
```

此外，还可以用关键字 **using** 引入单个名字，如 **using std::cout**。一旦引入了这个名字，后面就不需要再用名字限定了。

```
#include <iostream>
using std::cout;                          //引入 std 中的单个名字 cout
int main() {
    cout <<"hello world"<< std::endl;     //endl 必须名字限定
    return 0;
}
```

另外，输出运算符<<可以连续使用，这是因为 cout <<"hello world"的结果仍然是 cout，所以可以连续使用<<。

1.2.8 字符串和字符

用双引号括起来的如"hello world"是一个字符串，而用单引号''括起来表示一个字符，如'A'、'3'、'#'、'@'、'*'等。但一些特殊的字符，如那些表达特殊含义的字符，可以用一个**反斜杠字符**\加上另外一个字符来表示，如'**\n**'表示的不是 2 个字符而是一个叫作换行符的字符，输出运算符<<遇到这个换行符会执行换行动作，即输出另起一行。例如：

```
cout <<'h'<<'e'<<'l'<<'l'<<'o'<<'\n';
```

表示向 cout 输出 6 个字符，cout 对最后的换行字符'\n'执行一个换行动作，即从新的一行开始输出。

注：'\n'和 std::endl 都表示换行符，不过后者会强制程序的缓冲区里面的数据立即输出。

再如：

```
cout <<"hello\n\nworld";
```

这条语句将会在输出 hello 后，执行 2 个换行动作，然后输出 world。结果如下：

```
hello

world
```

还有其他的有特殊含义的转义字符，如'\t'表示制表符，输出运算符<<遇到这个字符会

输出几个空格。例如：

```
cout <<"hello\tworld";
```

这条语句将会在输出 hello 后，输出几个空格，然后输出 world。结果如下：

```
hello    world
```

通过 cout 用输出运算符<<输出一些字符或字符串，可在控制台窗口输出一些特殊的图案。如输出如下图案：

```
      *
    *   *
  *   *   *
*   *   *   *
```

程序代码如下：

```
#include <iostream>
using std::cout;
int main() {
    cout <<"       * "<<'\n';
    cout <<"     *   * "<<'\n';
    cout <<"   *   *   * "<<'\n';
    cout <<" *   *   *   * "<<'\n';
    return 0;
}
```

1.2.9 运算符和运算数

可以用运算符对运算数进行运算。如下面计算矩形的周长和面积的程序：

```
#include <iostream>
int main() {
    std::cout <<"长宽为"<< 5.8 <<','<< 3.4 <<"的矩形\n";
    std::cout << "其周长是: "<< 2 * (5.8 + 3.4) << '\n';
    std::cout << "其面积是: "<< 5.8 × 3.4 << std::endl;
    return 0;
}
```

可以看到，用＋或 * 运算符可以对数值进行加法和乘法运算，并且还可以用圆括号()来优先计算加法。当然，还有其他算术运算符，如减法运算符－、除法运算符/、求余数运算符%。

问题：

(1) 如果不用()，如何正确计算矩形的周长？

(2) 编写程序,测试对两个整数和 2 个实数的加、减、乘、除、求余数运算,看看结果是什么,有没有错误。

1.2.10 宏定义 #define

先看下面的计算不同半径的圆的面积的程序:

```
#include <iostream>
int main() {
    std::cout << 3.14 * 2.5 * 2.5 << std::endl;
    std::cout << 3.14 * 7.8 * 7.8 << std::endl;
    std::cout << 3.14 * 4.3 * 4.3 << std::endl;
    return 0;
}
```

上面程序的圆周率用 3.14 近似表示,但如果后来需要换用其他精度,如用 3.1415926 作为圆周率近似值,就需要在程序中搜索出所有的圆周率 3.14 并用 3.1415926 替换掉。

```
#include <iostream>
int main() {
    std::cout << 3.1425926 * 2.5 * 2.5 << std::endl;
    std::cout << 3.1415926 * 7.8 * 7.8 << std::endl;
    std::cout << 3.1415926 * 4.3 * 4.3 << std::endl;
    return 0;
}
```

假如程序中有很多地方用到圆周率,都需要这样找到它们然后替换,万一有一处忘记替换,结果将达不到预期的要求。一个更好的办法是给表示圆周率的数值起一个名字,如下所示:

```
#define PI 3.14
```

其中,以 # 开头的预处理指令 **#define** 称为宏定义,**#define PI 3.14** 给数值 3.14 起了一个名字,叫作 **PI**,这个 PI 就是一个宏。程序中用到 3.14 的地方可以用 PI 来代替。

```
#include <iostream>
#define PI 3.14
int main() {
    std::cout << PI * 2.5 * 2.5 << std::endl;
    std::cout << PI * 7.8 * 7.8 << std::endl;
    std::cout << PI * 4.3 * 4.3 << std::endl;
    return 0;
}
```

宏定义 PI 提高了程序的阅读性。更重要的是,如果今后需要更改圆周率,只要修改 #define PI **3.14** 为 #define PI **3.1415926** 即可,程序中其他地方无须修改。

1.2.11 变量

程序数据的值通常运行时才能确定，如前面的计算矩形面积的程序只能计算长、宽数值确定的矩形面积，无法计算其他长、宽数值的矩形面积。真正有用的程序要处理的具体数据值往往是不可预期的，通常来自外部输入(如键盘输入、文件、网络等)，如求任意矩形面积的程序应该可以根据外部输入(如键盘输入)计算相应的矩形面积。再如，一个程序用于统计分析某门课的成绩，程序员无法知道这个程序将用于什么班级，学生姓名、人数、考试成绩都无法预知。

也就是说程序的许多数据只有在程序运行时才能从外部获得，而不可能事先将数据"**硬编码**"在程序代码中。硬编码数据的程序只会产生固定的结果，不会根据不同的输入产生不同的结果，这样的程序是没什么用处的。

为了能接受外部输入的数据或保存计算过程中的中间结果，程序需要给它们准备相应的内存块。不同的数据需要不同的内存块，为了能区分这些不同的内存块，需要用不同的名字给这些内存块命名，程序可根据名字去访问这些内存块。这种内存块的名字称为**变量**。也就是说，**变量是命名的内存块**。

数据都有一个确定的数据类型，如 3 表示一个整数，而 3.14 表示一个浮点数(实数)。不同类型的数据占用的内存大小是不一样的，如字符'A'占据一个字节内存。一个整数通常占据 4 字节内存，而一个实数可能占据 8 字节内存。

同样，每个变量(在 C++中变量也称为对象)都有一个类型，用于定义它可以存储的数据的数据类型，编译器根据变量的类型给变量分配一定的内存。如：

```
int i;              //定义了一个叫作 i 的 int 型(int 整数类型)的变量
double r;           //定义了一个叫作 r 的 double 型(double 浮点实数类型)的变量
double area;        //定义了一个叫作 area 的 double 型的变量
```

上述代码定义了 3 个变量，它们分别是 3 个不同大小内存块的名字。当然，这些内存块的具体内容还是未定的，定义变量时，可以给变量对应的内存块存储一个初始值，例如：

```
int i{2};           //int 型(整型)变量 i 的值是 2
double r{2.5};      //double 型(浮点型)变量 r 的值是 2.5
double area = 0;    //double 型(浮点型)变量 area 的值是 0,可以用 = 代替{}设置初始值
```

即可以用花括号{}或=给这些变量对应的内存块一个初始值。3 个变量对应的内存块里分别存储了不同类型的数值：整数 2、浮点数 2.5 和浮点数 0。浮点数就是实数的意思。

下面的程序用变量 r 对应的内存块保存圆的半径值：

```
#include <iostream>
#define PI 3.14159
int main() {
    double r;           //r 是 double 类型的变量,double 是浮点实数类型
    r = 2.5;
    std::cout << PI * r * r << std::endl;
```

```
    //...
    return 0;
}
```

1.2.12 标准输入流对象 cin

前面程序中变量 r 的数值是硬编码的。可以修改程序，从键盘输入圆的半径到这个变量 r(实际是 r 对应的那个内存块)中。为此，需要用到标准输入输出流头文件中定义的叫作 **cin** 的输入流对象，它代表键盘对象。可以从 cin 中输入数据到某个程序变量(对象)中。

```
double r;
std::cin >> r;        //等待用户从键盘输入一个实数给变量 r。其中>>是输入运算符
```

上述代码从输入流对象 cin 中用输入运算符>>输入数据到变量 r 中。完整代码如下：

```
#include <iostream>
#define PI 3.14159
int main() {
    double r;
    std::cin >> r;    //等待用户从键盘输入一个实数给变量 r。其中>>是输入运算符
    std::cout << PI * r * r << std::endl;
    //...
    return 0;
}
```

程序执行到 std::cin >> r 时就会等待用户输入数据。当用户输入一个实数后，程序才会继续执行下一条语句。

1.2.13 用户定义类型

C++的变量都有一个数据类型，如前面表示整数的 int 类型、表示浮点数的 double 类型，这些都是语言自带的**内在数据类型**(简称**内在类型**)，此外，C++还允许程序员定义自己的数据类型(称为**用户定义类型**)。例如，标准库中的表示字符串的类型 string 就是一个用户定义类型。

```
#include <iostream>
#include <string>   //包含 string 类型的头文件
int main() {
    std::string s;    //string 也是属于标准名字空间 std 的名字
    std::cin >> s;
    std::cout <<"hello,"<< s << std::endl;
    return 0;
}
```

该程序中定义了一个 string 类型的对象，可以用输入运算符>>从输入流对象 cin 中输入一个字符串到这个变量中，也可以通过输出运算符<<输出这个对象到输出流对象 cout

中。执行该程序，输入一个字符串如 liping 后的执行过程如下：

```
liping
hello,liping
```

用户定义类型通常还定义一些成员函数和成员变量(即这些函数和变量是定义在这个类型中的)。例如，string 类型有一个 size()成员函数，可以返回一个 string 对象的字符个数，substr(s,e)成员函数返回 string 对象下标 s 到 e 之间的字符构成的一个字符串，甚至可以对 2 个字符串对象用运算符＋将它们拼接为一个新字符串。例如：

```
#include <iostream>         //std::cout
#include <string>           //std::string

int main (){
    std::string s("hello world");
    std::string s2 = s.substr(0,6);
    std::cout << s2 <<"  "<< s2.size()<< std::endl;
    std::cout << s + s2 << std::endl;
}
```

执行上述程序，结果如下：

```
hello 6
hello worldhello
```

1.3 数和字符的表示

1.3.1 数的表示

1. 十进制和二进制

程序的数据和代码在计算机内部都是以 0、1 串表示的。而在编程语言中，数的表示方式有多种。如可以用日常生活中常用的**十进制(decimal)**，即用 10 个不同的数字 0、1、2、……、9 表示一个数。对于小于或等于 10 的数可以直接用这 10 个不同的数字中的一个来表示或区分就可以了，即 1 位数就可以表示不超过 10 的数。但如果一个数超过了 10，就需要用 2 位数、3 位数等多位数来表示，即用**逢 10 进 1** 的多位数表示。如 11 表示的是 10＋1，整数 329 意思是“3 个 100”加上“2 个 10”加上“1 个 9”，可以表示成 10^i 的多项式：

$$329=3\times100+2\times10+9=3\times10^2+2\times10^1+9\times10^0$$

但在计算机中，由于计算机硬件即晶体管只有“开”和“关”2 种状态，也就是说一个晶体管开关只能区分 2 种不同情况，相当于只能表示 0 和 1 这 2 个数字。如何表示更大的数字？和十进制一样，可以采用“**逢 2 进 1**”的方法，采用多个晶体管，即用多个 0 和 1 的排列来表示一个很大的数。这种用 0 和 1 表示数的方法就称为**二进制**表示法。例如，二进制数 1011 可以表示成 2^i 的多项式：

$$1011=1\times2^3+0\times2^2+1\times2^1+1\times2^0=1\times8+0\times4+1\times2+1\times1=11$$

即相当于十进制数 11。

1 个二进制位经常也称为 1 比特(b),而 8 个二进制位经常也称为 1 字节(B)。表 1-1 所示是部分 8 位二进制数对应的十进制数。

表 1-1 部分 8 位二进制数对应的十进制数

二 进 制	十 进 制	二 进 制	十 进 制
0000 0000	0	1000 0000	128
0000 0001	1	1001 0001	145
0001 0001	17	1111 1111	255
0111 1111	127		

使用前 7 个二进制位可以表示 0～127 共 128 个不同数字,使用全部 8 位二进制位可以表示一共 256 即2^8个不同的数,用 n 位二进制位可以表示2^n个不同的数,如正整数 $0\sim2^n-1$。

在 C++语言中,用 0b 开头的一串二进制表示一个二进制数。如二进制数 1011 用 C++表示为:

```
0b1011
```

2. 十六进制

当处理更大的二进制数时会遇到一些问题,二进制位太多了,不方便。如:

```
1011 0101 1101 1001 1110 0101
```

表示成十进制数 11917797,只要 8 位数字。然而十进制数在某些情况下也不方便,如希望将从右往左的第 17 个二进制位设置成 1,就很难用十进制做到这点。一个解决方案是采用**十六进制**,即用 16 个不同的数来表示一个数。即用十进制的 10 个数 0、1、2、……、9 加上英文字母的前 6 个字母 A(a)、B(b)、C(c)、D(d)、E(e)、F(f)来表示一个数(其中的字母不区分大小写)。表 1-2 是十六进制、十进制和二进制之间的对应关系。

表 1-2 十六进制、十进制和二进制的对应关系

十 六 进 制	十 进 制	二 进 制
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A 或 a	10	1010

续表

十六进制	十 进 制	二 进 制
B或b	11	1011
C或c	12	1100
D或d	13	1101
E或e	14	1110
F或f	15	1111

因为一个十六进制数对应4个二进制数,所以可以将任何二进制数按照4个一组的方式以十六进制形式表示。如:

1011 0101 1101 1001 1110 0101

对应的十六进制表示法为:

b 5 d 9 e 5

对这种十六进制数,可以很容易地将其对应二进制的某一位(如第17位)设置为1。

也可以很容易地将十六进制采用如下方式计算出对应的十进制的值:

$$b\times16^5+5\times16^4+d\times16^3+9\times16^2+e\times16^1+5\times16^0$$

此外,还有八进制(octal),即用0、1、2、3、4、5、6、7表示一个数。

下列程序将二进制数0b101101011101100111100101以其他不同进制形式输出:

```
#include <iostream>
int main() {
std::cout <<"十进制: " << std::dec << 0b101101011101100111100101 <<'\n'
    << "十六进制: " << std::hex << 0b101101011101100111100101 <<'\n'
    << "八进制: " << std::oct << 0b101101011101100111100101 << '\n';
}
```

程序运行结果为:

```
十进制: 11917797
十六进制: b5d9e5
八进制: 55354745
```

其中,std::dec、std::hex、std::oct是控制输出形式的操纵符,分别表示以十进制、十六进制、八进制形式输出其后的数值。

1.3.2 字符的表示

在计算机中,各种字符,如大小写的英文字母,数字(0、1、2、……、9),一些特殊字符(如#、/、换行符、制表符等键盘上可见字符)也都是以二进制串表示的。例如整数值为35的二进制串表示字符#。不同的二进制串(对应一个整数)表示不同的字符,完全取决于怎么解释它们。

1. ASCII码

20世纪60年代,ASCII(American Standard Code for Information Interchange,美国信息变换标准编码)约定用7位二进制数表示128个不同的英文字符,但7位ASCII码对于法

国、德国等国家语言的字符就不够用了，于是提出了扩展的用8位二进制数表示字符的扩展ASCII字符编码。但8位ASCII编码无法表示中国、日本等国家语言的字(符)，如汉字总共有将近88 000个。为此，1990年人们提出了**统一字符编码**(Universal Character Set，UCS)。UCS是ISO 10646标准，UCS字符的编码长度为32位。

2. UCS和Unicode

UCS定义了字符和**编码点(code point)**的映射关系。即每个字符都对应一个确定整数值表示的编码点。编码点的范围不仅可以容纳所有语言的所有字符，还可以表示不同的图形符号，如数学符号，甚至是表情符号。

编码点和**编码(code)**并不是一回事，编码点是一个整数值，而编码是用一系列字节表示一个编码点的方式。不超过256的编码点用1字节就可以表示，如果用4字节而不是1字节存储这种编码点(值)就浪费了。因此，编码是有效地存储表示编码点(值)的方式。

Unicode是一个标准，不仅定义了一个字符集及其编码点(其中字符的编码点等同于UCS的对应字符的编码点)，还定义了多种表示编码点的**编码方式**。大多数语言字符都能用16位编码(code)表示。

Unicode提供了多种编码方法，但也造成了理解上的困惑，最广泛使用的Unicode编码方式包括**UTF-8**、**UTF-16**、**UTF-32**。其中每个编码方式都可以表示Unicode字符集中的所有字符。

- UTF-8表示一个字符采用的是变长的字节序列(即用1～4字节表示一个字符)，其中ASCII字符的编码使用1字节表示，和对应的ASCII编码是一样的。大多数网页的文本都采用UTF-8编码方式。
- UTF-16用1个或2个16位编码表示一个字符。UTF-16包含了UTF-8。
- UTF-32最简单，用一个32位编码表示所有的字符。

表1-3是Unicode编码点和UTF-8编码的关系。

表1-3 Unicode编码点和UTF-8编码的关系

Unicode编码点(十六进制形式)	UTF-8编码(二进制形式)
0000 0000-0000 007F	0xxxxxxx
0000 0080-0000 07FF	110xxxxx 10xxxxxx
0000 0800-0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

如汉字的“中”的十六进制表示是4E2D，其二进制形式是0100 1110　0010 1101，其Unicode的值位于第3行左列的范围内，因此，其UTF-8编码占用3字节，只要将其二进制位依次填入相应的x的位置就得到了其对应的UTF-8编码：1110**0100** 10**111000** 10**101101**。

表1-4是字符A和汉字“中”的Unicode编码点及UTF-8编码。

表1-4 字符A和汉字“中”的Unicode编码点及UTF-8编码

字符	ASCII	Unicode(十六进制)	Unicode(二进制)	UTF-8
A	01000001	0041	00000000 01000001	01000001
中	无	4E2D	01001110 00101101	11100100 10111000 10101101

1.4 编译、执行 C++程序

从 C++源代码创建可执行程序基本分为 3 个步骤。

第 1 步：预处理器(preprocessor)处理所有预处理指令。例如将所有＃include＜xxx＞预包含指令用文件 xxx 的内容替换，将程序中出现的用＃define NAME yyy 宏定义指令定义的名字 NAME 全部替换为 yyy。

第 2 步：编译器(compiler)将每个.cpp 文件编译为一个对应的目标文件，即将编程语言表示的语句代码(指令)都表示为二进制的机器代码(指令)。

第 3 步：连接器(linker)将程序的所有目标文件和依赖库的目标文件的二进制机器代码组合成一个二进制机器代码的可执行文件或库。

图 1-1 显示了由多个源文件组成的程序经过编译、连接得到最后的可执行文件的过程。一般的编译器通常都包含了预处理器，即运行编译命令时会自动调用预处理器对预处理指令进行处理，图 1-1 中省略了预处理过程。因此，一般执行编译器的编译(compile)就将一个源程序文件编译为二进制机器代码的目标文件，再通过连接器的连接(link)将这些目标文件和它们依赖的其他第三方(库)的目标文件组合成最终的可执行文件或库。

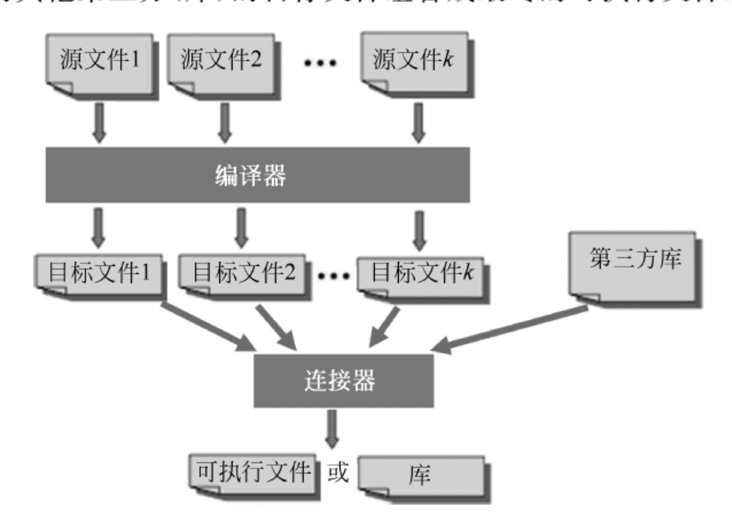


图 1-1　程序的编译(包含预处理过程)、连接过程

图 1-1 中的"库"，主要指的是一些他人已经编译好的二进制目标代码文件(即二进制库)。当然也有很多库是以源代码形式存在的，这些库中的源代码同样要经历预处理、编译、连接过程，并和程序的目标代码组合成一个可执行文件或二进制库。

不同的编译器编译、连接生成最终的可执行文件的过程和命令是不同的，许多编译器通过一个命令就能完成上面的 3 个步骤。如在 Linux/UNIX 平台上的 C++编译器 g++只要一个命令：

```
g++ hello.cpp
```

就能将一个源文件 hello.cpp 编译为一个可执行文件 a.out。这些编译命令通常还有一些选项，如-o 表示生成一个特定名字的可执行文件：

```
g++ hello.cpp -o hello
```

上述命令就生成了一个叫作 hello.out 而不是默认的 a.out 的可执行文件。

另外,不同的编译器对 C++标准的支持也是不同的,有的语言特征可能在某个编译器中是不支持的。下面的网址列出了著名的编译器对 C++17 标准的支持情况:https://zh.cppreference.com/w/cpp/compiler_support。

除了用命令行编译 C++程序外,有一些支持 C++语言的著名的集成开发环境(简称 IDE),如微软的 Visual Studio(针对 Windows 平台)、跨平台的 CodeBlocks 和 Clion 等。这些 IDE 集成了编辑源代码的文本编辑器、包含预处理编译链接的编译器、调试程序的调试器等,极大地方便了程序员,提高了程序的开发效率。

编程环境的安装和使用,请参考作者博客(https://a.hwdong.com)中的文章"C++17 安装、配置"或微软的 Visual Studio 2017 教程的文章,网址为 https://docs.microsoft.com/zh-cn/cpp/build/vscpp-step-1-create? view=vs-2017。

1.5 习题

1. 请编写程序输出下列图案。

```
A
B B
C C C
D D D D
E E E E E


     1
  2     3
4    5    6
```

2. 编写程序,让用户从键盘输入一个字符串和一个实数,程序再将这个字符串和实数分两行输出。

3. 补充下列代码,从键盘输入 2 个整数,计算它们的和、差、积、商、余数。

```
#include <iostream>
int main() {
    std::cout << "请输入 2 个整数: " << std::endl;
    int a,b;
    //...
    return 0;
}
```

4. 编写一个程序计算 cube(立方体)、sphere(球)和 cone(圆锥)的体积,计算公式如下。这些物体的参数可从键盘输入。

```
cubeVolume = side³
sphereVolume = (4/3) * pi * radius³
coneVolume = pi * radius² * (height/3)
```

5. 下列程序片段是否合法？如果不合法，则分析其原因并改正。

```
std::cin >> "hello world";
std::cout << "hello";
        << "world";
std::cout'hehe';
```

6. 下列程序片段有什么错误？为什么？

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* "*/"/*/;
std::cout << /* "*/"/**/";
```

7. 下列代码有什么错误，为什么？并改正。

```
#include <iostream>
int main() {
    string s;
    cin << s;
    cout << s;
}
```

8. 下列代码用关键字 namespace 定义了 A 和 B 两个名字空间。程序哪里有错误？请改正，并说明程序的输出是什么。

```
namespace A {
    void f() {
        std::cout << "A::f";
    }
    void g() {
        std::cout << "A::g";
    }
};
namespace B {
    void f() {
        std::cout << "B::f";
    }
}
using A::f;
int main() {
    f();
    B.f();
```

```
    g();
}
```

9. 将十进制整数 765431 转换为十六进制，在十六进制形式下，如何将该数的二进制的第 17 位设置为 1？

10. 请在网上查询一下换行符、回车符、制表符和汉字“汉”的 Unicode 编码点(值)，然后根据表 1-4 写出其对应的 UTF-8 的编码。

11. 网上查询 string 类型的 find()成员函数，并编写测试代码。



变量和类型

2.1 变量

变量就是命名的内存块，每个变量都具有确定的类型。定义变量就是给变量分配相应的内存块。

2.1.1 变量的定义及初始化

一个变量的定义格式是：

```
类型名　变量名　初始化式
```

即定义一个变量除了给变量一个名字外，必须说明这个变量的类型。可以不对变量初始化，也可以用不同的方式对变量初始化：

```
int a;
int b{};
int c{2};
int d = 2;
int e(2);
```

C++变量的初始化有很多种方式，上面是最常用的，其中，{}方式的初始化称为**列表初始化**，可以将变量的初始值放在这个花括号里，如果花括号里没有提供初始值，对于基本类型的变量，初始值将默认为 0，如上面的变量 b。有的编译器对没有提供初始化式的变量会发出警告或报错。

同一类型的多个变量可以在一个语句中定义，只要用逗号隔开这些变量的定义就可以了。如：

```
int main() {
    int a{}, b{}, c{ 2 }, d = 2, e(2);
```

```
    std::cout << a << ' ' << b << ' ' << c << ' ' << d << ' ' << e << '\n';
    return 0;
}
```

执行上述程序,输出是:

```
0 0 2 2 2
```

现代 C++建议采用{}这种列表初始化对变量初始化,对这种初始化,编译器会检查是否会导致**信息损失**。如:

```
int i1{3.5},i2 = {3.5},i3 = 3.5,i4(3.5);
```

用浮点类型的 3.5 对 int 型的变量初始化,会将 3.5 转换为 int 类型值 3,即小数部分被截取掉,导致了信息的损失。对于这种情况,编译器会报错,而其他方式则不报错,从而有助于防止对变量的错误初始化。

2.1.2 auto

从 C++11 开始就引入了 **auto** 关键字。用 auto 定义一个有初始值的变量时,不需要明确指定类型,因为编译器能自动从变量的初始值推断出该变量的数据类型。如:

```
auto b = true;      //true 是 bool(布尔)类型的一个值
auto ch{'x'};       //单引号表示的字符'x'是 char(字符)类型的值
auto i = 123;       //123 是 int(整数)类型的值
auto d{1.2};        //1.2 是 double(浮点)类型的值
auto z = d + i;     //从表达式 d + i 的值推断 z 的数据类型
```

2.1.3 typeid 运算符

可用 **typeid** 运算符查询得到一个数据类型或变量的类型信息。通过返回类型信息 type-info 对象的 name()成员函数(关于成员函数,会在第 7 章介绍)可得到这个数据类型的名字。

下列代码可查询 2.1.2 节代码中变量的类型信息。

```
cout << typeid(int).name() << '\t';
cout << typeid(b).name() << '\t';
cout << typeid(ch).name() << '\t';
cout << typeid(i).name() << '\t';
cout << typeid(d).name() << '\t';
cout << typeid(z).name() << endl;
```

这段代码输出的信息是:

```
int bool char int double double
```

其中,bool 是表示逻辑值的布尔类型,bool 类型只有 true(真)和 false(假)2 个值,而 char 是

字符类型，表示一个字符。

2.1.4 decltype

还可以用 decltype(exp)得到一个表达式的值的类型，并用这个类型来定义一个变量。如：

```
decltyp(3 + 4.5) c;
cout << typeid(c).name() << '\n';
```

将输出：

```
double
```

2.1.5 赋值运算符=

对于一个变量，可以用赋值运算符修改该变量的值(该变量内存的值)。例如，下面的程序输出为 5：

```
#include <iostream>
int main() {
    int a = 2;
    a = 5;
    std::cout << a << std::endl;
}
```

该程序定义了一个初始值是 2 的变量，然后用赋值运算符=将其修改为 5。注意，定义变量时的=符号不是赋值运算符。

2.1.6 const

可以用 const 关键字修饰一个变量(对象)，用来表示变量的**不可修改性**。如：

```
const int i = 3;    //const 表示变量 i 是不可被修改的
i = 4;              //错：试图修改 const 对象
```

因为 const 定义的变量 i 是不能被修改的，因此在定义这个变量时就必须初始化，如果不初始化，也是错误的。如：

```
const int j;        //错：j 没有初始化
```

报告的错误为：

```
error C2734: "j": 如果不是外部的，则必须初始化常量对象
```

2.1.7 标识符、关键字、文字量

变量名如 a、i1 等都是一些标识符。所谓**标识符**就是一些由字母、数字和下画线组成的且不能以数字开头的字符串。变量名、函数名等都是标识符。如 i1 是由字母和数字组成的标识符，而下面程序中表示圆半径的变量 radius 是完全由字母组成的标识符。

```
#include <iostream>
int main() {
    double radius;
    std::cin >> radius;     //等待用户从键盘输入一个实数给变量 r

    std::cout << 3.1415926 * radius * radius << std::endl;
    return 0;
}
```

同样，表示数据类型的 int、double 也是一些标识符，return 也是标识符，它们是被 C++ 语言已经使用的**关键字**（也称为**保留字**）。程序员自己定义的变量名、函数名等不能与 C++ 的关键字相同。

上面程序中，变量 radius 的数值来自用户的输入，是会变化的。而 3.1415926 是一个直接给出确定数值的量，称为**"文字量"**。可以看到一个程序中的数据分为变量和文字量，变量对应一块内存，而文字量直接编码在代码中，没有独立的可寻址的内存。

2.2 数据类型

C++是一个静态类型语言，所有数据都必须具有确定的数据类型。数据类型（简称类型）规定了数据的内容是什么样的（是整数、实数还是字符），该类型对象占据内存大小是多少，对该类型的数据能进行什么运算。如 int 类型表示的整数通常占据 4 字节，表示的整数范围是−32767～32767，对 int 类型的数据可以进行整数的加、减、乘、除、取模（求余数）等运算。double 类型的对象通常占据 8 字节内存，可以进行实数的加、减、乘、除，但不能进行取模（求余数）等运算。例如：

```
double x = 3.5, y(4);
std::cout << x % y;
```

编译器编译时，将产生编译错误：

```
error C2296: "%": 非法,左操作数包含"double"类型
```

即编译器会在编译程序时会根据变量类型为变量分配合适大小的内存，根据变量类型检查是否支持相应的操作（运算），从而自动帮助我们发现程序中的 bug。

C++语言本身已经定义了基本类型（如 int、char、double）和复合类型（数组、指针、引用），这些数据类型称为**内在类型**。C++还允许程序员定义自己的数据类型，这种类型称为

用户定义类型，如第 1 章中表示字符串的 string 类型。本章主要介绍 C++的基本类型。

2.2.1 基本类型

基本类型又分为算术类型和 void 类型。其中算术类型又包括整型和浮点类型。

整型又分为以下几种。

- 布尔类型：bool 类型，只有 true 和 false 2 个值，用于表示真和假。
- 整数类型：包括基本的整数类型 int、short、long、long long 以及和表示正负数的修饰符 signed(有符号的)、unsigned(无符号的)结合而产生的整数类型。例如，int 和 signed、unsigned 结合产生的整数类型 signed int 和 unsigned int。
- 字符类型：包括 signed char、unsigned char、char、utf-8、wchar_t、char16_t、char32_t。

字符类型分为 signed(符号)和 unsigned(无符号)，signed char 表示的字符对应的整数包括正整数和负整数，而 unsigned char 是非负整数。char 根据目标平台和编译器不同，可能等价于 unsigned char 也可能等价于 signed char。wchar_t 表示宽字符，可表示任何字符编码点(在支持 Unicode 的系统上为 32 位，但在 Windows 系统上为 16 位，即相当于 UTF-16)。char16_t 表示 UTF-16 字符，即占据 16 位的字符。char32_t 表示 UTF-32 字符，即占据 32 位的字符。

int 类型如果和其他修饰符结合使用，则可以省略关键字 int。修饰符包括表示符号性(**signedness**)的修饰符(unsigned 和 signed)、表示大小(**size**)的修饰符(short、long)。表 2-1 是各种整数类型及其在不同数据模型中的内存大小。

表 2-1 各种整数类型及其在不同数据模型中的内存大小

<table>
<tr><th rowspan="2">类型说明符</th><th rowspan="2">等价类型</th><th colspan="5">数据模型的位宽</th></tr>
<tr><th>C++ standard</th><th>LP32</th><th>ILP32</th><th>LLP64</th><th>LP64</th></tr>
<tr><td>short</td><td rowspan="4">short int</td><td rowspan="6">至少 16</td><td rowspan="6">16</td><td rowspan="6">16</td><td rowspan="6">16</td><td rowspan="6">16</td></tr>
<tr><td>short int</td></tr>
<tr><td>signed short</td></tr>
<tr><td>signed short int</td></tr>
<tr><td>unsigned short</td><td rowspan="2">unsigned short int</td></tr>
<tr><td>unsigned short int</td></tr>
<tr><td>int</td><td rowspan="3">int</td><td rowspan="5">至少 16</td><td rowspan="5">16</td><td rowspan="5">32</td><td rowspan="5">32</td><td rowspan="5">32</td></tr>
<tr><td>signed</td></tr>
<tr><td>signed int</td></tr>
<tr><td>unsigned</td><td rowspan="2">unsigned int</td></tr>
<tr><td>unsigned int</td></tr>
<tr><td>long</td><td rowspan="4">long int</td><td rowspan="6">至少 32</td><td rowspan="6">32</td><td rowspan="6">32</td><td rowspan="6">32</td><td rowspan="6">64</td></tr>
<tr><td>long int</td></tr>
<tr><td>signed long</td></tr>
<tr><td>signed long int</td></tr>
<tr><td>unsigned long</td><td rowspan="2">unsigned long int</td></tr>
<tr><td>unsigned long int</td></tr>
</table>

续表

类型说明符	等价类型	数据模型的位宽				
		C++ standard	LP32	ILP32	LLP64	LP64
long long	long long int (C++11)	至少 64	64	64	64	64
long long int						
signed long long						
signed long long int						
unsigned long long	unsigned long long int (C++11)					
unsigned long long int						

浮点类型分为 float、double、long double，即单精度浮点数、双精度浮点数、长双精度浮点数。IEEE 754(https://en.wikipedia.org/wiki/IEEE_754)标准定义了浮点数的表示格式，包括−0，反常值(denormal number)，特殊值(如 inf 表示无穷、nan 表示非数值)。一个浮点数由符号位、指数、尾数组成。例如 32 位 float 类型浮点数 $b_1b_2b_3\cdots b_9b_{10}b_{11}\cdots b_{32}$ 一般可表示为：$(-1)^s * 2^{e-127} * 1.f$。其中 $s=b_1$ 表示符号的 1 位二进制，$e=b_2b_3\cdots b_9$ 是 2 的指数部分的 8 位二进制，因为 e 的范围是[0,255]，所以 2^{e-127} 表示的指数范围是 $[2^{0-127}, 2^{255-127}]=[2^{-127}, 2^{128}]$，$f=b_{10}b_{11}\cdots b_{32}$ 是尾数的 23 位二进制。一般情况下，尾数 1.f 是介于 1 和 2 之间的，这种浮点数是正规的(normal)，但对于特别小如接近于 0 的数，可以让尾数小于 1，这样的浮点数是子正规的(subnormal)。IEEE 754 对 64 位浮点类型 double 的表示是类似的。long double(长双精度浮点数)不一定映射到 IEEE 754 规定的类型，通常是 x86 和 x86-64 架构上的 80 位 x87 浮点类型，内存大小通常也是 64 位，但表示精度不同于 double。

不同类型的数据表示的数值的范围是不同的，如表 2-2 所示。

表 2-2 不同类型的数据表示的数值的范围

类型	位长	格式	值范围	
			近似	精确
字符	16	signed		−128～127
		unsigned		0～255
	16	unsigned		0～65 535
	32	unsigned		0～1 114 111 (0x10ffff)
整型	16	signed	$-3.27\times10^4\sim3.27\times10^4$	−32 768～32 767
		unsigned	$0\sim6.55\times10^4$	0～65 535
	32	signed	$-2.14\times10^9\sim2.14\times10^9$	−2 147 483 648～2 147 483 647
		unsigned	$0\sim4.29\times10^9$	0～4 294 967 295
	32	signed (2 的补码)	$-9.22\times10^{18}\sim9.22\times10^{18}$	−9 223 372 036 854 775 808～9 223 372 036 854 775 807
		unsigned	$0\sim1.84\times10^{19}$	0～18 446 744 073 709 551 615

续表

类型	位长	格式	值　范　围	
			近　　似	精　　确
浮点	32	IEEE 754	最小子正规： $\pm\ 1.401\,298\,4\times10^{-45}$ 最小正规： $\pm\ 1.175\,494\,3\times10^{-38}$ 最大： $\pm\ 3.402\,823\,4\times10^{38}$	最小子正规：±0x1p−149 最小正规：±0x1p−126 最大：±0x1.fffffep+127
	64	IEEE 754	最小子正规： $\pm\ 4.940\,656\,458\,412\times10^{-324}$ 最小正规： $\pm\ 2.225\,073\,858\,507\,201\,4\times10^{-308}$ 最大： $\pm\ 1.797\,693\,134\,862\,315\,7\times10^{308}$	最小子正规：±0x1p−1074 最小正规：±0x1p−1022 最大：±0x1.fffffffffffffp+1023

浮点数只能表示绝对值在一定范围内的实数，其中的最大表示绝对值最大的正数和负数，而最小表示绝对值最小的正数和负数。例如对于32位正规浮点数，其表示的实数近似范围是$[-3.402\,823\,4\times10^{38}, -1.175\,494\,3\times10^{-38}]$和$[1.175\,494\,3\times10^{-38}, 3.402\,823\,4\times10^{38}]$。

参考网址：https://en.cppreference.com/w/cpp/language/types。

2.2.2　sizeof 运算符

可以用C++的sizeof运算符来查询一个变量或类型在不同操作系统和编译器环境下实际占用的内存的大小。如：

```
#include <iostream>
int main() {
    int i = 3;
    char ch = 'A';
    double radius = 2.56;
    bool ok = false;

    std::cout <<"int 整型占用内存：" << sizeof(int) <<"字节"  << std::endl;
    std::cout <<"i 占用内存：" << sizeof(i)  <<"字节"  << std::endl;
    std::cout <<"ch 占用内存："  << sizeof(ch) <<"字节"  << std::endl;
    std::cout <<"radius 占用内存："<< sizeof(radius) <<"字节" << std::endl;
    std::cout <<"ok 占用内存：" << sizeof(ok)  <<"字节"  << std::endl;
}
```

将输出如下信息：

```
int 整型占用内存：4 字节
i 占用内存：4 字节
```

```
ch 占用内存: 1 字节
radius 占用内存: 8 字节
ok 占用内存: 1 字节
```

2.2.3 文字量

直接写出值的常量如 2、3.14、'X'、"hello world"等都称为**文字量**,上面的文字量分别是int(整)型、double(浮点)型、char(字符)型和 string(字符串)文字量。也就说每个文字量也有对应的数据类型。

1. 整型文字量

整型文字量可以表示为十进制、八进制、十六进制、二进制等不同形式。以 0 开头的是八进制,以 0x 或 0X 开头的是十六进制,以 0b 或 0B 开头的是二进制。例如,整数 18 的不同进制文字量如下:

```
18   022   0x12   0b100010010
```

可以用 typeid()查询这些量的数据类型:

```
cout << typeid(18).name() << ' '
    << typeid(022).name() << ' '
    << typeid(0x12).name() << ' '
    << typeid(0b100010010).name() << '\n';
```

上述语句将输出:

```
int int int int
```

为了表示其他的整型,可以在文字量后面添加表示不同整型的后缀,字母 u 或 U 表示 unsigned 整型,字母 l 或 L 表示 long 整型,ll 或 LL 表示 long long 整型。如:

```
18u   022L   18LL   0x12UL   18ULL
```

同样,可以用下列语句输出这些文字量的类型:

```
cout << typeid(18u).name() << '\n'
    << typeid(022L).name() << '\n'
    << typeid(18LL).name() << '\n'
    << typeid(0x12UL).name() << '\n'
    << typeid(18ULL).name() << '\n';
```

输出:

```
unsigned int
long
__int64
```

```
unsigned long
unsigned __int64
```

定义整型变量时，如果指定了变量的具体类型，则可以省略文字量后缀。如：

```
unsigned long a{18};
unsigned short b {18};
long long c {18};
```

如果初始值超出了变量类型的取值范围，则结果不可预期。如：

```
unsigned char ch{512u};
unsigned int i{ - 1};
```

unsigned char 的值的范围是[0,255]，unsigned int 不可能是负数。用{}列表初始化，编译器(VS2017)会报告语法错误：

```
error C2397: 从"unsigned int"转换到"unsigned char"需要收缩转换
error C2397: 从"int"转换到"unsigned int"需要收缩转换
```

2. 浮点型文字量

浮点型文字量必须包含小数点，可用后缀 f 或 F 表示 float、用 l 或 L 表示 long double 浮点类型。如：

```
3.14   3.14f   3.14 F 3.14L   3.14l
```

还可以在浮点型文字量后面用 e 加一个整型文字量，表示 10 的指数形式的浮点数。如：

```
2E3    0.2e - 3    - 0.1E - 3L    .3E2f
```

执行下列语句：

```
cout << 2E3 << '\t' << 0.2e - 3 << '\t'
    << - 0.1E - 3L << '\t'<< .3E2f <<'\n';
```

输出：

```
2000.0     0.0002    - 0.0001   30
```

3. 字符(串)文字量

单引号表示单个字符，双引号表示一个字符串，字符串中可以有 0 到多个字符。如：

```
'A'、"hello"、"X"、""
```

其中,第 1 个是字符'A';后面 3 个表示字符串,"X"是只有一个字符的字符串,""是不包含任何字符的空字符串。注意,'A'和"X"属于完全不同的数据类型。'A'的数据类型是 char。

如果要表示不同类型的字符类型,可以用前缀。如:

```
L'A'   u'A'   U'A'   u8'A'
```

其中,L、u、U 分别是 wchar_t、char16_t、char32_t。u8 前缀实际是用于表示 UTF-8 字符串的。如:

```
u8"你好,liping"
```

表示将字符串中的字符(包括汉字字符)都用 UTF-8 编码方案进行编码。

4. 转义字符序列

如何在字符串中表示双引号字符?如何表示哪些不可见的字符,如空字符、换行符、制表符?如何表示特殊字符,如响铃符?解决方法:用反斜杠开始的转义字符序列表示某种字符,如\n 表示换行符,\t 表示制表符,\0 表示空字符(结束符)等。

所有的 ASCII 字符都可以用反斜杠\和其 8 位 ASCII 表示。如:

\0(空字符)　　\7 (响铃)　　\12 (换行符)　　\40(空格)

\115 (‘M’)　　\x4d (‘M’)

\后的值不能超过 256,因此,一般不超过 3 位十进制数。如\1234 表示的是字符\123和'4',\402 是非法的。

表 2-3 是一些常见字符的 ASCII 值及其转义字符。

表 2-3　一些常见字符的 ASCII 值及其转义字符

转义字符	ASCII 值	表示的字符	含义或作用
\0	0	空字符	字符串结束
\n	12	换行符	换到新行
\t	9	水平制表符	空几个空格
\v	11	垂直制表符	
\r	13	回车	回到开头
\\	92	\	
\'	39	'	
\''	34	''	双引号
\?	63	?	问号
\a	7	响铃符	响铃
\xhh	2 位十六进制		

有时,需要用原始字符串而不需要处理转义字符,如将\n 看成单独的 2 个字符而不是一个转义字符,可用 R 开头的字符串表示,其格式为:

```
R"delimiter(raw_characters)delimiter"
```

其中，delimiter 是除圆括号()、反斜杠和空格字符之外的字符序列。如：

```
std::cout << R"d2f(\\hello\rwor\01d)d2f";
```

输出：

```
\\hello\rwor\01d
```

2.2.4 格式化输出

可以用流操纵符（stream manipulators）控制数据的输出格式。这些流操纵符定义在 2 个头文件（iomanip 和 ios）中。可以用输出运算符<<将一个操纵符作用于输出流对象，如 cout。例如：

```
bool b{true};
std::cout << b << '\t';
std::cout << boolalpha << b << std::endl;
```

输出：

```
1      true
```

cout 默认将 bool 型变量的值默认转换为整数 1 或 0 输出，如果在前面插入了流操纵符 boolalpha，就以字符串"true"或"false"的形式输出 bool 型变量的值。

ios 头文件已经自动被 iostream 头文件包含，该头文件中操纵符不带任何参数，例如可以用如下操纵符将整数以特定进制格式输出。

std::dec：后续的整数都以十进制形式输出。

std::hex：后续的整数都以十六进制形式输出。

std::oct：后续的整数都以八进制形式输出。

如：

```
std::cout << std::hex << 18 <<'\t'<< 25 << '\n';
std::cout << std::oct << 18 << '\t'<< 25 << '\n';
std::cout << std::dec << 18 << '\t'<< 25 << '\n';
```

输出：

```
12        19
22        31
18        25
```

可以用如下操纵符改变浮点数的输出格式。

std::fixed：以固定精度形式输出。

std::scientific：以科学计数法形式输出。

std::hexfloat：以十六进制浮点形式输出。

std::defaultfloat：以默认形式输出。

如：

```
std::cout   << std::fixed << 0.01 << '\n'
        << std::scientific << 0.01 << '\n'
        << std::hexfloat << 0.01 << '\n'
        << std::defaultfloat << 0.01 << '\n';
```

输出：

```
0.010000
1.000000e-02
0x1.47ae14p-7
0.01
```

而 iomanip 的操纵符往往需要传递一个参数，如 setw(5)操纵符表示输出量占据的宽度是 5。常用的操纵符如下。

std::setw(n)：改变输出域的宽度。

std:: setprecision(n)：改变浮点数的精度。

std:: setfill(ch)：改变填空字符，当 setw 的输出域宽度大于输出值的宽度时，默认的填空字符是空格，可以用 setfill(ch)改变这个填空字符。

如：

```
std::cout << std::setprecision(2) << 3.1415926 << '\n';
std::cout << std::setw(10) << 3.1415926 << '\n';
std::cout << std::setw(10) << std::setfill('-') << 3.1415926 << '\n';
```

输出：

```
3.1
       3.1
-------3.1
```

2.2.5 类型转换

1. 隐式类型转换

用运算符对不同类型的数据进行运算或定义变量的初始化值类型和变量类型不一致时，C++编译器会将自动它们转换为同一种类型，称为隐式类型转换。例如：

```
int  i = 3.14;          //在对 int 型变量 i 初始化时，会将
                        //double 型值 3.14 转换为 int 型值 3，再对 i 初始化
int  j = i * 2.5;       //i 先转换为和 2.5 同类型的 double 值 3.0，2 个
                        //double 值相乘的结果再转换为 int 类型值，对变量 j 初始化
```

2. 强制类型转换

除了隐式类型转换外，有时需要强制将一种类型值转换为另一种类型值。一种方法是用C语言的旧式强制类型转换：

```
(T) var
```

通过在变量var前的圆括号()将变量var强制转换为()里的类型T的值。如：

```
double c = 2/(double)5;
```

将int类型值5强制转换为double型值，然后和int型值2相除，此时会自动将2隐式转换为double型值再进行double值的除法运算。

C++还提供了一个**static_cast<T>**运算符可以将一个变量强制转换为<>之间指定的数据类型T的值。如上述语句也可以写成：

```
double d = 2/ static_cast<double>5;
```

下面是一个包含隐式和强制类型转换的例子。

```
#include <iostream>
using namespace std;
int main() {
    bool b = 42;            //int 型值 42 转换为 bool 值 true,再对 b 初始化,b 的值就是 true
    int i = b;              //b 的值 true 转换为 int 型值 1,再初始化 int 变量 i
    std::cout << boolalpha << b <<'\t'<< i << std::endl;
    i = 3.14;               //double 型值 3.14 转换为 int 型值 3,再对变量 i 赋值,i 的值就是 3
    std::cout << i << std::endl;
    unsigned char c = -1;       //unsigned char 的取值范围是[0,255]
                                //-1 关于[0,255]的余数为(-1+256) % 256 = 255
    signed char c2 = 256;       //256 超出 char 的取值范围[-127,128],结果不可知
    std::cout << (unsigned short)c << std::endl;
    std::cout << (short)c2 << std::endl;
    return 0;
}
```

输出：

```
true    1
3
255
0
```

3. unsigned 类型

混合int和unsigned类型时，会用取模法（余数法）将负整数隐式转换为unsigned类型的值。

```
int main(){
    unsigned u = 0;
    u = u-1;                  //如果 unsigned 是 32 位整数:[0, 4294967295]
                              //-1 转换为:(-1+4294967296) % 4294967296
    std::cout << u << std::endl;
    int i = -42;
    cout << i + i << endl;    //输出 -84
     cout << u + i << endl;   //如果是 32 位整数,则输出 4294967264
                         //-42 转换为:(-42+4294967296)%4294967296 = 4294967254
}
```

输出：

```
4294967295
-84
4294967253
```

2.2.6 类型别名

可以用关键字 using 给一个数据类型起另外的名字，称为**类型别名**。如：

```
using FLOAT = double;
```

给数据类型 double 起了一个别名 FLOAT。这个 FLOAT 类型就是 double 类型，定义 FLOAT 类型的变量就是 double 类型的变量。如：

```
FLOAT radius{2.5};             //相当于 double radius{2.5};
```

在 C++11 之前的传统 C++标准中，可以用另外一个关键字 typedef 定义等价的类型别名：

```
typedef double FLOAT;
```

定义别名有两个用途，一个用途是使代码具有移植性。同一个数据类型的数据在不同平台占据内存的大小和表示值的范围是不一样的，可以通过定义数据类型别名，使得自己代码中的数据在任何平台都是一样的，只要针对每个平台用数据类型别名将数据的类型定义为一致的数据类型就可以了。例如：

```
#if defined USING_COMPILER_A
using Int32 = __int32;
using Int64 =  __int64;
#elif defined USING_COMPILER_B
using Int32 = int;
using Int64 = long long;
#endif
```

这里的 #if 和 #endif 是**条件预处理语句**，预处理器如果发现定义了代表某种编译器的宏 USING_COMPILER_A 就用上面的两个 using 语句，否则就用下面的两个 using 语句。

另外一个用途是提高代码的可读性和紧凑型。例如：

```
typedef std::basic_string<char> string;
std::basic_string<char> s("hello");
std::string   s2("world");
```

用 typedef 给类型 std::basic_string<char>起了一个简单的名字 string，使得定义该类型的变量 s2 比 s 更加简化和更具可读性。

2.2.7　枚举

除了 C++ 自带的内在数据类型外，程序员还可以定义自己的数据类型。如可以用关键字 enum class 定义一个枚举数据类型：

```
enum class Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
Day d{Day::Tuesday };
```

上述代码定义了一个叫作 Day 的数据类型。所谓枚举就是这个类型在定义时就列举出了这个类型所有可能的值，如这个 Day 类型的变量只有 7 个可能的值，d 是初始化为 Day 类型的值 Day::Tuesday。

不同枚举类型的值是不能相互比较或赋值的。

```
enum class Color { red, green, blue };
enum class Color2 { red, green, blue,yellow };
int main() {
    Color c = Color::green;
    Color2 c2 = Color2::red;
    c2 = c;
}
```

产生如下编译错误：

```
error C2440: '=': cannot convert from 'Color' to 'Color2'
```

2.3　局部变量与全局变量、变量的作用域与生命期

2.3.1　程序块、局部变量和全局变量

用{}包围的一组语句称为**程序块**。在一个程序块内部定义的变量称为**局部变量**(也称为**内部变量**)，反之，称为**全局变量**(也称为**外部变量**)。

对于基本类型的变量，作为外部变量定义时如果没有给初始值，则其值默认为 0；而作

为内部变量定义时如果没有给初始值，则其值是不确定的。如：

```
int g_var;
int main() {
    int var;
    std::cout << g_var << '\t'<< var << '\n';
}
```

VS 编译器对于局部变量 var 会报告错误：

```
error C4700: 使用了未初始化的局部变量"var"
```

也就是说，如果没有初始化局部变量，则其值是不确定的。给它一个初始化：

```
int var{2};
```

输出结果：

```
0       2
```

2.3.2 作用域和生命期

每个变量都有一个作用域(scope)和生命期，全局变量的作用域是整个程序，即程序中从它出现后的任何地方都可以访问它，并且直到程序结束，全局变量才销毁，即它的生命期就是整个程序的生命期。而局部变量的作用域是从它定义的地方开始，直到它所在的程序块的结尾，超出它所在的程序块，这个变量就不存在了，当然就不能访问。因此，它定义所在的程序块就是它的**作用域**。

```
int g_var;
int main() {
    int var{2};
    {
        std::cout << g_var << '\t'<< var << '\n';
        int var{ 0 };
        var = 5;
        std::cout << g_var << '\t'<< var << '\n';
    }
    std::cout << g_var << '\t'<< var << '\n';
}
```

该程序有一个属于 main()函数程序块的局部变量 var，还有一个属于其内部{}程序块的局部变量 var。第二个 var 的作用域就是这个内部{}程序块，超出这个程序块，该局部变量 var 就不存在(销毁)了。尽管对内部的 var 赋值 5，最后 main()函数输出的 var 值仍然是 2。

另外，内部程序块的变量会隐藏外部程序块的同名变量，如内部{}程序块中的 var 就隐藏了 main()程序块中的 var，即在内部{}程序块的 var 变量出现后的地方，如果访问 var 则

都是指的内部{}程序块的 var 变量而不是 main()程序块中的 var 变量，因此内部第二个输出语句的 var 值是 5，而第一条输出语句的 var 就是 main()程序块中的 var 值 2。程序运行结果如下：

```
0       2
0       5
0       2
```

2.4 习题

1. 下面哪些是不合法的变量标识符？为什么？

@ohd　　*zara　　a2bc　　move_name　　a_123

myname50　　_temp　　j　　a23b9　　retVal　　51_name

2. 编写程序，将从键盘输入的两个字符串用+运算符连接成一个字符串，然后输出这个拼接的新字符串。

3. 编译下面的程序，看看有什么编译错误。

```
#include <iostream>
#include <string>
int main() {
    std::string s = "hello", s2 = "world";
    std::cout << s * s2;
    std::cout << 3 * s ;
}
```

4. 指出下面文字量的数据类型。

A. 'a', L'a', "a", L"a",u'a', U'a'

B. 10, 10u, 10L, 10uL, 012, 0xC,0xAuLL

C. 3.14, 3.14f, 3.14L, .314e-2L

D. 10, 10u, 10., 10e-2

5. 下列程序的输出是什么？

(1)

```
int main(){
    int a = -1,b = -1;
    unsigned c = 1;
    cout << a * b << endl << a * c << endl;
}
```

(2)

```
int main(){
    unsigned u = 42,u2 = 10;
```

```
    cout << u - u2 << endl;
    cout << u2 - u << endl;
    int i = 10, i2 = 42;
    cout << i - i2 << endl;
    cout << i2 - i << endl;
    cout << i - u << endl;
    cout << u - i << endl;
}
```

(3)

```
int main(){
    for(unsigned int i = 10; i>=0;i--)
      cout << i << endl;
}
```

6. 完善下面的程序，根据输入字符串的值设置相应的 Color 类型变量 color 的值。

```
#include <iostream>
#include <string>
using namespace std;
enum class Color { red, green, blue };
int main() {
    string str;
    Color color;
    cin >> str;
    if (str == "red") color = Color::red;
    //...
}
```

7. 输入 3 个整数，按照从小到大的顺序输出。如输入 9,1,3，输出为 1,3,9。

8. 对 bool 类型的值能否进行加、减、乘、除运算？

9. 下列程序的输出是多少？为什么？

```
#include <iostream>
int a;
int main(){
    int b;
    std::cout << a << '\t' << b << '\n';
}
```

10. 编写一个程序，按照类似下列格式，输出你的操作系统中各种常用基本类型的变量占用内存的大小。

```
int 型占用内存空间大小:     4 字节
char 型占用内存空间大小:    1 字节
```

11. 修改下列代码的语法错误，运行修改后的程序，体会变量的不同初始化方式。

```
#include <iostream>
int main() {
    double d;
    double d1{ 3.5 };
    double d2 = { 3.5 };
    double d3 = 3.5;
    double d4(3.5);
    int i;
    int i1{ 3.5 };
    int i2 = { 3.5 };
    int i3 = 3.5;
    int i4(3.5);

    std::cout << d << '\t'<< d1 << '\t'
        << d2 << '\t'<< d3 << '\t'<< d4 << '\n';

    std::cout << i << '\t'<< i1 << '\t'
        << i2 << '\t'<< i3 << '\t'<< i4 << '\n';

    auto a = 3.5;
    auto a1 { 3.5 };
    auto a2 = i2 + d2 / 2;
    std::cout << a << '\t'<< a1 << '\t'<< a2 << '\n';
}
```

12. 下面两组变量的定义有什么区别？有没有错误？为什么？

(1) int month{ 8 }, day{ 6 };

(2) int month{08},day{ 06 }。

13. 下列关于 auto 的用法哪里有错误？为什么？

```
auto x;
auto y{};
auto z{ 0 };
auto u = z;
auto v(u);
```

14. 下面程序的输出是什么？

```
#include <iostream>
int main() {
    int a = 0;
    decltype((a)) b = a;
    b++;
    std::cout << a <<'\t'<< b;
}
```



运算符与表达式

3.1 运算符

运算符是对数据进行数学或逻辑操作的特殊符号，如运算符＋、＊、||分别表示加、乘、逻辑或。运算符对数据进行运算就构成了表达式。

3.1.1 运算符的分类

根据功能的不同，运算符可分为算术、比较、逻辑、位、赋值等，如表 3-1 所示。

表 3-1 运算符的功能分类

功　能	运　算　符
算术	＋、－、＊、/、%(求余数)、＋＋(自增 1)、－－(自减 1)
比较	>、<、>=、<=、!=(不等于)
逻辑	&&(逻辑与)、||(逻辑或)、!(逻辑非)
位	&(与)、|(或)、^(异或)、~(补)
赋值	=、+=、-=、*=、/=、%=、&=、^=

根据参与运算的运算数的个数，运算符可分为一元、二元、三元运算符。

如加法运算符＋需要 2 个运算数，称为**二元运算符**。而自增运算符++，只使一个运算数自己增加 1。如：

```
int a = 2;
++a;                    //a 的值从 2 变成了 3,即自增了 1,即相当于 a = a+1
std::cout << a;         //将输出 3
```

这种只对一个运算数进行运算的运算符叫作**一元运算符**。

三元运算符只有一个，即条件运算符?:。其格式为：

```
exp1?exp2:exp3
```

该条件运算符对3个运算数即3个表达式exp1、exp2、exp3进行条件运算。整个表达式的值根据exp1的值是否为true(真)或非0而取exp2或exp3的值。即如果exp1的值是true(真)或非0值,整个表达式的值就是exp2的值,否则是exp3的值。

例如,假设要求a、b两个数的最小值,可以这样:

```
a<b? a : b
```

如果a<b是true,则该表达式值是a,否则是b。即该表达式返回a和b的最小值。

假如要求3个数a、b、c的最小值,可以用下面的语句:

```
a<b ? (a<c?a:c): (b<c?b:c)
```

即如果a<b成立,则在a和c中继续用条件运算符比较,否则在b和c中继续用条件运算符比较。

可以用一小段代码测试一下上述代码:

```
#include<iostream>
int main(){
    int a,b,c;
    std::cin>>a>>b>>c;
    int min = a<b ? (a<c?a:c): (b<c?b:c);
    std::cout<<a<<","<<b<<","<<c<<"这3个数的最小值是: "
    <<min<<std::endl;
}
```

从键盘输入3个整数(输入时,中间用空格隔开),看看结果是否正确。

3.1.2 优先级和结合性

不同运算符具有不同的优先计算次序,如*、/优先于+、-。下面的表达式:

```
x+y*z
```

先计算*(乘法),后计算+(加法)。

C++中每个运算符都具有一个优先级,如*、/的优先级都是5,而+、-的优先级都是6,判断两个量是否相等的运算符==的优先级是10,即优先级数字小的运算符更优先。

不需要死记这些运算符的优先级,可以用圆括号()来保证正确的计算次序。如(x+y)*z将先计算括号里面的后计算括号外面的。

表达式中有多个相邻的相同运算符时,这多个相同运算符的计算次序取决于该运算符的结合性是自左向右还是自右向左。加法运算符+是自左向右计算的,如x+y+z先计算左边的+,其结果再用于右边的+运算,而赋值运算符=是自右向左计算的,如x=y=z的计算过程是:先y=z(z的赋值给y),然后x=y(即表达式y=z的结果值y赋值给x)。

C++运算符的优先级和结合性可以参考如下网址：https://en.cppreference.com/w/cpp/language/operator_precedence。

3.2 表达式

运算符对数据(变量和文字量)运算构成**表达式**。即表达式是由0个以上运算符和1个以上运算数构成的。如表达式2+3由2个文字量2和3通过加法运算符+构成。

最简单的表达式仅由1个变量或文字量构成，不含任何运算符。如上述表达式中的3本身也是一个表达式。

每个表达式都有运算结果，对它们可以继续运算。如表达式2+3是由表达式2和表达式3通过加法运算符+构成的表达式。因此，表达式中可以嵌套其他表达式。

再看下面的例子：

```
z = x + y/2 + x * z
```

其中，x、y、z、2都是表达式，y/2和x * z也是表达式，x+y/2也是表达式，x+y/2+x * z也是表达式，z=x+y/2+x * z通过赋值运算符=构成表达式。

上述表达式的计算次序可以用圆括号表示如下：

```
(z = ((x + (y/2)) + (x * z))  )
```

当然，可以用()改变计算次序：

```
z = ((x + y)/2  +  x) * z)
```

3.3 算术运算符

如表3-2所示，二元算术运算符有+、-、*、/、%。其中%是“求余数”(也称取模)运算符，用来求2个整数相除的余数。

表3-2 二元算术运算符的含义及示例

运算符	含义	例子	结果
+	加	15+2	17
-	减	2-15	-13
*	乘	15 * 2	30
/	除	15/2	7
%	求余数	15%2	1

此外，还有对一个数进行运算的一元运算符：自增运算符++、自减运算符--、正号运算符+、负号运算符-。假设一个变量a的值是2，这些一元运算符的含义及示例如表3-3所示。

表 3-3 一元算术运算符的含义及示例

运 算 符	含 义	例 子	例 子 解 释
++	后置自增	a++	表达式的值是 a 原先的值 2，然后 a 增加 1，变为 3
	前置自增	++a	a 先增加 1，表达式的值为增加后的 a 值 3
−−	后置自减	a−−	表达式的值是 a 原先的值 2，然后 a 减少 1，变为 1
	前置自减	−−a	a 先减去 1，表达式的值为减少后的 a 值 1
+	正号	+a	+a 就是 a 自身
−	负号	−a	a 的相反数

3.3.1 算术运算符需要注意的几个问题

1. 溢出问题

每种类型的变量在内存中占据一定的空间，其表示值的范围也不同，如 short 类型值占 2 字节（16 位），其最高位表示正负号，因为 $2^{15}-1=32\ 767$，short 可表示的整数范围是 [−32 767，32 767]。如果值超出了该类型的表示范围，则结果是不可预期的，即产生了溢出（overflow）。如：

```
int main(){
    short value = 32767;
    std::cout << value << std::endl;
    value = value + 1;    //value 超出 short 类型的值的范围，结果不可预期
    std::cout << value << std::endl;
}
```

2. 整数相除/

除法运算符/对两个整数运算的结果总是整数，如果不能整除，则结果会截取掉小数部分。

```
auto val = 21/6;     //结果是 3
auto val2 = 21/7;    //结果是 3
auto val3 = 21.0/6;  //double 型数的实数相除，结果是 3.5
```

再如：

```
21/6;               //结果是 3
21/7;               //结果是 3
-21/(-8);           //结果是 2
21/(-5);            //结果是 -4
```

3. 求余数运算 %

求余数运算%只能用于两个整数，不能用于浮点数。如：

```
int ival = 42;
```

```
double dval = 3.14;
auto val = ival % 4;
auto val2 = dval % 4; //错: % 不能用于实数
```

整数的求余运算 % 的结果的符号和被除数相同，即 m%n 的符号和 m 的符号相同。如：

```
21 % 6;             //结果是 3
21 % 7;             //结果是 0
-21 % (-8);         //结果是 -5
21 % (-5);          //结果是 1
```

4. 整数和浮点数混合运算

当运算符对整数和浮点数运算时，会将整数转换成浮点数，再进行运算。如：

```
auto val3 = 21.0/6;
```

上述代码中，6 隐含从 int 类型转换为 double 类型，再按 doule 类型进行运算，结果是 3.5。

如果参与运算的量都是整型，仍然希望它们按照浮点型进行运算，可以对变量进行显式的强制类型转换。可以用 C++的 **static_cast** <>对变量进行强制类型转换，格式如下：

```
static_cast<T> (var);
```

上述代码将变量 var 强制转换为类型 T 的值。

```
int main(){
    int a = 3, b = 4;
    std::cout <<"int/int = :"<< a/b << std::endl;
    std::cout <<"double/int = :"<< static_cast<double>(a)/b << std::endl;
    std::cout <<"int/double = :"<< a/static_cast<double>(b)<< std::endl;
    std::cout <<"double/double = :"<< static_cast<double>(a)/static_cast<double>(b)<< std
    ::endl;
}
```

上述代码中的后 3 个输出语句的结果是一样的。因为当 int 和 double 类型值混合运算时，会自动将 int 类型值转换为 double 类型值再运算。

3.3.2 自增++和自减--

自增++分为前置++和后置++。前置和后置的区别如下。

- ++x：先运算后结果，即先对 x 增加 1，再将 x 的值作为表达式的值。
- x++：先结果后运算，即表达式的值是未运算前的 x 值，然后 x 自身增加 1。

看一个例子：

```
#include <iostream>
int main(){
    int a = 3, b = 3;
    std::cout << "a++的值"<< a++<<", a 的值是"<< a << std::endl;
    std::cout << "++b 的值"<<++b <<", b 的值是"<< b << std::endl;
}
```

输出结果：

```
a++的值 3, a 的值是 4
++b 的值 4, b 的值是 4
```

看一个复杂一点的自增++的例子。

```
#include <iostream>
using namespace std;
int main(){
    int x = 1;
    int a = ++x;      //x 先增加 1 为 2,表达式++x 为更新后的 x 值 2,a 的值为 2
    int b = x++;      //表达式 x++的值为 x 的值 2,即 b = 2,然后 x 自增 1 为 3
    int c = ++ ++x;   //c 和 x 的都是 5
    int d = x+ ++x;   //因为 x 和++x 是 + 的 2 个运算数,但 x 和++x 哪个先计算,值是不确定的
                      //结果可能是 5 + 6,也可能是 6 + 6
    int e = x++ ++;   //出现编译错误:需要左值
    cout << a << endl << b << endl << c << endl
    << d << endl << e << endl;
    return 0;
}
```

x++ ++会出现编译错误：需要左值。这是因为表达式 x++的值是“先取 x 的值，然后 x 增加了 1”。也就是表达式 x++的结果不是 x 本身而是存在一个临时变量中，所以不能对临时变量再应用++运算。

实际编程中，应该避免这种连续使用两个++的情况。

3.3.3　数学计算函数库 cmath

对于一些复杂的运算，如求平方根或求指数运算，C++没有相应的运算符，需要借助于 C++的函数库，如从 C 语言继承来的数学函数库（头文件是 cmath）来运算，程序中需要用 #include 将数学库的头文件 cmath 包含进来。

例如求一个实数的根，需要借助于 cmath 的 sqrt()函数来得到一个实数的平方根。

```
#include <cmath>
#include <iostream>
int main(){
    double d = 3.5;
    std::cout << sqrt(3.5)<< std::endl;     //输出 3.5 的平方根
```

```
    return 0;
}
```

cmath 还有其他数学函数，如 pow(a,b)用于求 a 的 b 次方。如：

```
#include <cmath>
#include <iostream>
int main(){
    double base = 3.5,exp  = 6.4;;
    std::cout << base <<"^"<< exp <<"等于"<< pow(base,exp)<< std::endl;
    return 0;
}
```

表 3-4 是其中的一些常用的数学函数，这些函数接受浮点型或整型的参数，返回的是浮点型的结果。

表 3-4 常用的数学函数

函数名	含义
abs(x)	绝对值函数：返回 x 的绝对值。注意，x 如果是整数，返回的也是整数。如 abs(−2)的结果是整数 2。abs(−2.0)的结果是 2.0
ceil(x)	天花板函数：返回大于或等于 x 的最小整数对应的浮点值。如 ceil(−2.5)的结果是−2.0，ceil(2.6)的结果是 3
floor(x)	地板函数：返回小于或等于 x 的最大整数对应的浮点值。如 ceil(−2.5)的结果是−3.0，ceil(2.6)的结果是 2
round(x)	返回最接近 x 的整数对应的浮点值。如 round(−2.3)的结果是−2.0，而 round(−2.6)的结果是−3
pow(x,y)	底数是 x 指数是 y 的指数函数的值。如 pow(2.3,4.5)
exp(x)	底数是自然数，指数是 x 的指数函数的值。如 exp(2.3)
log(x)	底数是自然数，值是 x 的对数函数的值
log10(x)	底数是 10，值是 x 的对数函数的值
sqrt(x)	平方根函数，如 sqrt(2)的值是 1.414

此外，cmath 还包含了各种三角函数和反三角函数，如 sin()、tan()、acos()、atan()等。

三角函数接受的是弧度值，如果是角度，需要转换成弧度，同样，反三角函数的返回的也是弧度而不是角度。如：

```
#include <cmath>
#include <iostream>
int main() {
    float angle_deg{ 60.0f } ;        //角度
    const float pi{ 3.14159265f };
    const float pi_degrees{ 180.0f };
    float tangent{ std::tan(pi * angle_deg / pi_degrees) };
    float angle(std::atan(tangent));
    float angle_deg2{ angle * pi_degrees/pi };
```

```
    std::cout << angle << '\t'<< angle_deg2 << '\t'<< angle_deg << '\n';
}
```

输出：

```
1.0472  60      60
```

3.4 位运算

任何量在计算机内存里都是用一串二进制位表示的，假如有 2 个整数 a= 37 和 b = 22 可用 1 字节即 8 位表示，其内存的二进制位如下所示：

```
a = 0 0 1 0 0 1 0 1
b = 0 0 0 1 0 1 1 0
```

如果想输出一个数的二进制形式，可以用头文件 bitset 里的 bitset 模板转换为二进制形式：

```
#include <iostream>
#include <bitset>
#include <iomanip>
int main() {
    char a{ 37 }, b{ 22 };
    std::cout << std::setw(5)<<"a:"<< std::bitset<8>(a) << '\n'
        << std::setw(5) << "b:"<< std::bitset<8>(b) << '\n';
}
```

输出结果：

```
a:00100101
b:00010110
```

bitset<8>的箭头<>之间的 8 表示以 8 位二进制形式输出。对于占 4 字节(32 位)的 int 型值，可以用 bitset<32>转换为 32 位二进制。如：

```
std::cout << std::bitset<32>(a) << '\n';
```

输出结果：

```
00000000000000000000000000100101
```

位运算对运算数的二进制位进行相应的运算。其中 &(位与)、|(位或)、^(异或)是对 2 个运算数的对应二进制位进行运算，而～(补运算)、<<(左移位)、>>(右移位)则是对一个运算数的二进制位进行运算。

假设 p、q 分别是 a、b 两个数的对应位置的二进制位，表 3-5 是二元位运算符 &、|、^对 p 和 q 的运算规则。

表 3-5 二元位运算符的运算规则

p	q	p&q	p|q	p ^q
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

对于运算符 &，只有当 p 和 q 都是 1 时，p&q 的结果才是 1，否则都是 0。

对于运算符|，只要 p 和 q 有一个是 1 时，p|q 的结果就是 1，当两者都是 0 时，结果才是 0。

对于运算符^，只有当 p 和 q 不同(一个是 1，一个是 0)时，p ^q 的结果才是 1，否则都是 0。

因此，对 a、b 的 &、|、^运算结果如下：

```
a&b = 0 0 0 0 0 1 0 0
a|b = 0 0 1 1 0 1 1 1
a ^b = 0 0 1 1 0 0 1 1
```

一元位运算符～(补运算)、<<(左移)、>>(右移)的运算规则如下。

- ～(补运算)：返回一个数的补，即对 x 的每一位取反。结果相当于－x－1。
- <<(左移)：各二进位全部左移若干位，高位丢弃，低位补 0。
- >>(右移)：各二进位全部右移若干位，若为无符号数，则高位补 0；若为负数，则高位补 1。

例如，对 b 的<<、～运算的结果如下：

```
b<<2 = 0 1 0 1 1 0 0 0
~b = 1 1 1 0 1 0 0 1
```

运行下列程序：

```
#include <iostream>
#include <bitset>

int main() {
    char a{ 37 }, b{ 22 };
    std::cout <<"a:"<< '\t'<<(short)a <<'\t'<< std::bitset<8>(a) << '\n'
        << "b:"<< '\t'<<(short)b <<'\t'<< std::bitset<8>(b) << '\n';

    std::cout << "a&b" << '\t'<< (a&b) << '\t'<< std::bitset<8>(a&b) << '\n';
    std::cout << "a|b" << '\t'<< (a|b) << '\t'<< std::bitset<8>(a|b)  << '\n';
    std::cout << "a ^b" << '\t'<< (a ^b) << '\t'<< std::bitset<8>(a ^b)  << '\n';

    std::cout << "~a" << '\t'<< (~a) << '\t'<< std::bitset<8>(~a)<< '\n';
```

```
    std::cout << "a << 2" << '\t'<< (a << 2)<< '\t'<< std::bitset < 8 >(a << 2) << '\n';
    std::cout << "a >> 2" << '\t'<< (a >> 2) << '\t'<< std::bitset < 8 >(a >> 2) << '\n';
}
```

输出结果：

```
a:      37      00100101
b:      22      00010110
a&b     4       00000100
a|b     55      00110111
a^b     51      00110011
~a      -38     11011010
a<<2    148     10010100
a>>2    9       00001001
```

3.5 赋值运算符

最基本的**赋值运算符**是＝。如 a＝b 就是将 b 的值赋值给变量 a，即将 b 的值复制到变量 a 的内存块中。基本赋值运算符＝和算术运算符或位运算符组合构成了**复合赋值运算符**，如赋值运算符＋＝的运算 a＋＝b 就相当于 a ＝ a＋b，即将 b 的值加到 a 上去。表 3-6 是各种赋值运算符。

表 3-6　各种赋值运算符

运　算　符	例　　子	含　　义
＝	a＝b	将 b 的值赋给变量 a
＋＝	a＋＝b	相当于 a ＝ a＋b
－＝	a－＝b	相当于 a ＝ a－b
*＝	a *＝b	相当于 a ＝ a * b
/＝	a/＝b	相当于 a ＝ a/b
%＝	a%＝b	相当于 a ＝ a%b
&＝	a&＝b	相当于 a ＝ a&b
|＝	a|＝b	相当于 a ＝ a|b
^＝	a ^＝b	相当于 a ＝ a ^b
<<＝	a<<＝2	相当于 a ＝ a<<2
>>＝	a>>＝2	相当于 a ＝ a>>2

关于赋值运算符的说明如下。

- 右结合性，赋值运算符按从右到左的次序计算。如 a＝b＝c 是先计算 b＝c，然后计算 a＝b。
- 赋值运算符右运算数的类型应和左边算数的类型一致或能隐式转换为左运算数的类型。
- 赋值运算符的左边必须是一个左值（可修改的具有独立内存的变量），不能是文字量或 const 变量，也不能是表达式。如 34＝a 或 a＋b＝c 都是错误的。

3.6 关系运算符

关系运算符也称比较运算符，就是比较两个量的大小或判断是否相等。如比较两个量是否相等的==、不相等的!=运算符，比较一个量是否小于另一个量的<、小于或等于的<=，等等。

关系运算符的运算结果是一个 bool 型的值 true 或 false。

看一个具体的程序例子：

```
#include <iostream>
using namespace std;
int main(){
    int a = 4,b = 5;
    bool b1 = a<b;          //bool b1 = (a<b)
    bool b2 = a==b;         //bool b2 = (a==b)
    //boolalpha使得后面的bool变量输出"true"或"false"而不是1或0
    cout << boolalpha << b1 << '\t'  << b2 << endl;
}
```

输出结果：

```
true    false
```

表 3-7 列出了关系运算符及其含义。

表 3-7　关系运算符及其含义

运　算　符	例　　子	含　　义
==	a==b	a、b 相等时，返回 true，否则返回 false
!=	a!=b	a、b 不等时，返回 true，否则返回 false
<	a < b	a 小于 b 时，返回 true，否则返回 false
<=	a <= b	a 小于或等于 b 时，返回 true，否则返回 false
>	a > b	a 大于 b 时，返回 true，否则返回 false
>=	a >= b	a 大于或等于 b 时，返回 true，否则返回 false

注意：对于 2 个浮点数，不能用==判断它们是否相等，如下列程序。

```
#include <iostream>
using namespace std;
int main(){
    double d1(100 - 99.99);
    double d2(10 - 9.99);
    bool b = d1 == d2;          //bool b = (d1 == d2)
    cout << boolalpha << b << endl;
}
```

输出结果：

```
false
```

尽管理论上 d1 和 d2 都应该是 0.01，但由于计算机不能精确表示浮点数(只能近似)，因此，d1 和 d2 在计算机内部的表示并不是准确的理论值，是有误差的。执行下列程序：

```
#include <iostream>
#include <iomainp>
using namespace std;
int main(){
    double d1(100 - 99.99);
    double d2(10 - 9.99);
    bool b = d1 == d2;                //bool b = (d1 == d2)
    cout << boolalpha  << b << endl;
    cout << setprecision(17);         //浮点数输出格式修改为精度 17 位
    cout << d1 << endl;
    cout << d2 << endl;
}
```

输出结果：

```
false
0.010000000000005116
0.0099999999999997868
```

因此，判断 2 个浮点数是否相等，通常是看它们差的绝对值是否足够小。如：

```
#include <iostream>
#include <cmath>                      //绝对值函数 fabs()在 cmath 头文件中
#include <iomainp>
using namespace std;
int main(){
    double d1(100 - 99.99);
    double d2(10 - 9.99);
    bool b = fabs(d1 - d2)< 1e - 10;  //误差是否足够小
    cout << boolalpha  << b << endl;
    cout << setprecision(17);
    cout << d1 << endl;
    cout << d2 << endl;
}
```

输出结果：

```
true
0.010000000000005116
0.0099999999999997868
```

3.7 逻辑运算符

如表 3-8 所示,有 &&、||、! 共 3 种逻辑运算符,对 bool 类型的量进行运算。

表 3-8 逻辑运算符

运算符	含义	运算规则
&&	与	2 个都是 true 或 0 值时,结果才为 true
||	或	有 1 个是 true 或非 0 值时,结果就为 true
!	非	一元运算符,true 或非 0 值变为 false,false 或 0 值变为 true

看如下程序:

```
#include <iostream>
int main() {
    int a = 4, b = 0;
    std::cout << std::boolalpha;
    std::cout << (a || b) << std::endl
        << (a&&b) << std::endl
        << (!a&&b) << std::endl;
}
```

程序的结果为:

```
true
false
false
```

注意:参与逻辑运算的量如果不是 bool 类型,通常会隐式转换为 bool 类型的值,非 0 值或非空值会转换成 true,0 或空值会转换成 false。

逻辑运算符 && 和||具有一种短路特性。

- &&:只有当左操作数为真时才去检查右操作数。
- ||:只有当左操作数为假时才去检查右操作数。

C++逻辑运算符!、&&、||分别有等价的关键字 not、and、or。例如 a&&b 也可以写成 a and b。

3.8 特殊运算符

3.8.1 条件运算符

3.2 节中已经介绍过条件运算符?:,它是一个三元运算符,其格式为:

```
exp1 ? exp2 : exp3
```

当 exp1 是 true 时，结果为 exp2 的值，否则是 exp3 的值。

3.8.2 逗号运算符

格式为：

```
exp1, exp2
```

依次计算 exp1 和 exp2 的值，整个表达式的值是 exp2 的值。

注意：逗号运算符优先级低于赋值运算符。因此，语句“z=a,b;”等价于“(z=a),b;”。即表达式“z=a,b”是由表达式 z=a 和 b 构成的逗号表达式，即先计算表达式 z=a，然后再计算表达式 b，整个表达式的值是 b 的值。而“z=(a,b)”是先计算逗号表达式(a,b)，然后其结果即 b 的值赋给变量 z。如：

```
#include <iostream>
int main() {
    auto a{ 4 }, b{ 3 };
    //auto c = a, b;                //错：逗号运算符号优先级低于 = ,无法自动推进 b 的类型
    auto d = (a, b);
    std::cout << d << std::endl;
}
```

3.9 习题

1. 编写程序，输入一元二次方程的系数 a、b、c，输出该方程的 2 个根。

```
#include <cmath>
#include <iostream>
int main(){
    double a,b,c;
    //补充代码
}
```

2. 输入一个正整数，要求输出数字是逆序的正整数。如输入 2357，应输出 7532。

3. 输入一个整数，判定其是否是回文。所谓回文，就是其逆序的整数和原来的整数是同一个整数。如 12321 其逆序仍然是 12321，而 1231 则不是回文。

4. 下列运算的结果是什么？

```
unsigned long ul1 = 3, ul2 = 7;
```

A. ul1 & ul2　　B. ul1 | ul2　　C. ul1 && ul2　　D. ul1 || ul2

5. 下列运算的结果是什么？

A. (true&&true)||false　　B. (false&&true)||true

C. (false&&true)||true||true　　D. (5>6||4>3)&&(7>8)

E. !(7>4||3>4)

6. 下列语句的输出是什么?

```
std::cout << (2, 3.4 , 5.5);
```

7. 下列程序的输出是什么?

```
#include <iostream>
int main() {
    int a = 0;
    int b;
    a = (a == (a == 1));
    std::cout << a;
    return 0;
}
```

A. 0　　B. 1　　C. 很大的负数　　D. −1

8. 下列程序的输出是什么?

```
# include <iostream>
int main() {
    int x = 10;
    int y = 20;
    x += y += 10;
    std::cout << x <<" "<< y;
}
```

A. 40 20　　B. 40 30　　C. 30 30　　D. 30 40

9. 下列代码的作用是什么?

```
x = x | 1 << n;
```

A. 将 x 设置为 2^n　　B. 设置 x 的第 n+1 位为 1

C. x 的第 n+1 位取反　　D. 设置 x 的第 n+1 位为 0

10. 下面程序的输出是什么?

```
#include <iostream>
int main() {
    int x = 10;
    int y = (x++, x++, x++);
    std::cout << x <<" "<< y;
}
```

A. 13 12　　B. 13 13　　C. 10 10　　D. 依赖于编译器

11. 下列代码片段的输出是什么？为什么？

```
auto x{ 6 }, y{ 8 };
auto c{ false }, d{ true };
std::cout << (c ? ++x, y++ : --x, y--) << '\n';
std::cout << (true ? ++x, ++y : --x, --y) << '\n';
std::cout << c ? ++x, ++y : --x, --y << '\n';
```

12. 编写程序，输出下面 3 个值：−10、−10≫1、−10>2 的二进制形式。



语　　句

4.1 简单语句、复合语句和控制语句

4.1.1 简单语句

C++的大部分语句都以分号结尾。最简单的语句是只有一个分号的**空语句**。

```
;
```

还有变量定义语句：

```
int ival = 3, jval;          //定义了 2 个 int 类型变量 ival,jval
auto radius = 2.15;          //定义了 radius 的变量,其类型通过 auto 自动推断
```

由表达式加分号构成的表达式语句是常见的简单语句。

注意：表达式有一个结果，即表达式的值，而表达式语句则没有结果。如 std::cout << ival 是一个表达式，其结果是 std::cout 本身，而"std::cout << ival;"是一个表达式语句，没有结果。但该语句执行一个指令动作，即在控制台窗口输出 ival 的值。

4.1.2 复合语句

用花括号{ }括起来的一系列语句构成一个**复合语句**（也称为**程序块**或**语句块**）。如：

```
#include <iostream>
int main(){
    auto a = 1,b = 0;
    {
        a = 3;
        auto b = 5;
    }
```

```
    std::cout << a <<'\t'<< b << std::end;
}
```

上述程序中首先主函数体是花括号{ }包围的一个复合语句，其中又包含一个用{}括起来的复合语句，即：

```
{
    a = 3;
    auto b = 5;
}
```

main()函数体复合语句中有 2 个局部变量 a、b，main()函数内部嵌套的复合语句内部也有一个局部变量 b。当离开这个内部复合语句时，其局部变量 b 就销毁不存在了。因此，最后数据语句中的 a、b 都是 main()函数体复合语句中的局部变量。因此程序的输出是：

```
3 0
```

程序块通常包含多条语句，也可以只有一条语句。当只有一条语句时，可以省略{}，因此，简单语句也可以认为是一个程序块。

4.1.3 控制语句

前面的程序都是由简单语句和复合语句构成的，程序的执行按照它们定义的先后次序依次执行，但有时希望某些语句只在特定条件下才执行，或者希望某些语句在特定条件下重复执行，即某些语句会根据条件是否满足而决定是否执行其他语句。这样的语句称为**控制语句**。

C++的控制语句主要分为**条件语句**、**循环语句**(也称为**迭代语句**)和**跳转语句**。

4.2 条件语句

条件语句分为 **if** 语句和 **switch** 语句。

4.2.1 if 语句

用 if 关键字定义的 if 语句的基本格式是：

```
if(表达式)
   程序块
```

英文 if 的含义是“如果”。上述 if 语句的含义是：如果表达式的值是 true(或者是能转换为 true 的非 0 值)，就执行其中的程序块；反之，就不会执行其中的程序块。

例如：

```
double score;
std::cin >> score;
```

```
if(score < 60)
      std::cout <<"不及格!"<< std::endl;
```

当输入的分数小于60时,才执行输出语句。

if语句还有另外一种形式:

```
if(表达式)
    程序块1
else
    程序块2
```

其含义是如果表达式的值是true,则执行程序块1,否则执行程序块2。例如:

```
double score;
std::cin >> score;
if(score < 60)
    std::cout <<"不及格!"<< std::endl;
else
    std::cout <<"及格了!"<< std::endl;
```

if语句还可以**嵌套使用**。即一个if语句里可以包含其他的if语句。例如:

```
if(表达式)
    程序块1
else
    if(表达式2)
        程序块2
```

因为多个空格不影响语句的意思,所以可以写成更紧凑的形式:

```
if(表达式)
    程序块1
else if(表达式2)
    程序块2
```

或者

```
if(表达式)
    程序块1
else if(表达式2)
    程序块2
else
    程序块3
```

n个不同的条件的形式为:

```
if(表达式)
    程序块1
```

```
else if(表达式 2)
    程序块 2
else if(表达式 3)
    程序块 3
//...
else
    程序块 n
```

例如：

```
double score;
std::cin>>score;
if(score<60)
    std::cout<<"不及格!"<<std::endl;
else if(score<70)
    std::cout<<"及格!"<<std::endl;
else if(score<80)
    std::cout<<"中等!"<<std::endl;
else if(score<90)
    std::cout<<"良好!"<<std::endl;
else
    std::cout<<"优秀!"<<std::endl;
```

在使用 if 嵌套语句时，需要注意 if-else 的匹配是从内到外的。例如：

```
double score;
std::cin>>score;
if(score>=60)
    if(score>90)
        std::cout<<"优秀!"<<std::endl;
else
    std::cout<<"不及格!"<<std::endl;
```

尽管在写法上 else 子句似乎和外层的 if 子句对齐，但实际上 else 是和里面的 if 先匹配，构成一个完整的 if-else 语句并作为外部 if 语句的语句块。即实际上这段代码相当于：

```
double score;
std::cin>>score;
if(score>=60)
    if(score>90)
        std::cout<<"优秀!"<<std::endl;
    else
        std::cout<<"不及格!"<<std::endl;
```

显然，这不符合程序的本来意图，因为 60～90 分都被判为“不及格”了。

为了表示正确的程序设计意图，可以借助于{}来控制 if 和 else 的匹配，即可以写成：

```
double score;
std::cin>>score;
```

```
if(score>=60){
    if(score>90)
        std::cout<<"优秀!"<<std::endl;
}
else
    std::cout<<"不及格!"<<std::endl;
```

当一个 if 或 else 块里有多条语句时，也要用花括号{}括起来，不然其含义就不对了。如：

```
#include<iostream>
int main() {
    auto a{0};
    std::cin>>a;
    if (a<0)
        std::cout << "a=" << a << std::endl;
        std::cout << "这是一个负数"  << std::endl;
}
```

执行程序，输出结果：

```
8
这是一个负数
```

这个 if 语句体实际只有一条输出语句，因此，不管输入的是否是负数，都会执行最后一个输出语句。正确做法是用{}包围 2 条输出语句，构成 if 子句的程序块。

下列程序代码有什么错误？请改正。

```
int a = 100;
if(a<0)
    std::cout<<"a 的绝对值是";
    std::cout<<-a;
else
    std::cout<<"a 的绝对值是";
    std::cout<<a;
```

4.2.2 switch 语句

switch 语句格式是：

```
switch(可无损转换为整型的表达式){
    case (整型或枚举型)常量表达式 1:
        程序块 1
    case (整型或枚举型)常量表达式 2:
        程序块 2
    //...
    default:
```

```
        默认程序块
}
```

根据 **switch** 后圆括号()内的表达式是否等于某个 **case** 中的常量表达式,决定是否执行这个 **case** 中的程序块,如果没有相等的 **case**,则执行 **default**(默认情形)的程序块。如:

```
int x;
std::cin >> x;
switch(x){
    case 0:
    case 1:
        std::cout <<"x 是 0 或 1\n";
        break;                  //break 关键字用于跳出 switch
    case 2:
        std::cout <<"x 是 2\n";
        break;                  //break 关键字用于跳出 switch
    default:
        std::cout <<"x 不是 0,1,2\n";
        break;                  //break 关键字用于跳出 switch
}
```

根据 x 匹配哪一个 case,跳到哪个 case 子句执行,遇到 **break** 则跳出 switch 语句,否则会一直执行下去。当输入的是 0 或 1 时,则执行“std::cout <<"x 是 0 或 1\n";”,然后跳出 switch 语句。当输入的不是 0、1、2 等其他整数时,则执行 default 的默认语句“std::cout <<"x 不是 0,1,2\n"”。

下面程序可以用于统计从键盘输入的元音字符和非元音字符的个数:

```
unsigned vowelCnt = 0, nonVowelCnt = 0;
char ch;
while(std::cin >> ch){
    switch(ch){
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            vowelCnt++;
            break;              //跳出 switch 循环
        default:
            nonVowelCnt++;
            break;              //跳出 switch 循环
    }
}
```

关于 switch 语句有几个语法点:

- case 的标签必须是整型常量表达式。
- 不同的 case 标签值不能相同。

- case 子句里定义变量时必须加花括号，否则执行其他 case 时会出现“作用域里的该变量定义未初始化”。
- default 标签可以省略。
- break 关键字定义的 break 语句用于跳出整个 switch 语句。

下面的代码片段说明了 switch 语句的一些常见错误及其原因。

```
//下面的 switch 省略了 default 标签
int x,y;
switch(x){
    case 3.14:                //错：case 标签必须是整型常量表达式
        //... do something
        break;
    case y:                   //错：case 标签必须是整型常量表达式
        //... do something
        break;
}
```

上述代码 case 标签中的不是整型常量表达式。而下面代码片段中出现了 2 个 case 具有相同的整型常量值：

```
int x;
//...
//下面的有 2 个 case 的标签值都是 1,不能相同
switch(x){
    case 1:
        /...
        break;
    case 1:
        //...
        break;
    case 2:
        //...
        break;
}
```

再看下面的代码片段：

```
switch (v) {
case 1: int x = 0;            //初始化
      std::cout << x << '\n';
      break;
default:                      //编译错误：因为 default 标签,可能导致 'x' 未初始化
      std::cout << "default\n";
      break;
}
```

变量 x 可能因为直接跳到 default 而跳过 case 1 从而导致变量 x 未被初始化。因此，如果一个 case 需要定义局部变量，应该将变量定义在{}的程序块里，如下所示：

```
switch (v) {
    case 1: {   int x = 0;
    std::cout << x << '\n';
    break;
    }                                    //'x'的作用域在此结束
default: std::cout << "default\n";       //无错误
    break;
}
```

4.2.3 if/switch 语句中的初始化语句

C++17 还允许在 if/switch 语句中添加一个初始化语句。其格式如下：

```
if(初始化语句;表达式)
switch(初始化语句;表达式)
```

假如有下面一段代码：

```
{
    auto var = doSomething();
    if(condition(var))
        //if 块
    else
        //else 块
}
```

在 C++17 中可以写成：

```
if(auto var = doSomething;condition(var))
    //if 块
else
    //else 块
```

这有两个好处：一个是代码更加简洁；另一个是 var 只属于 if 语句，从而不会污染周围环境。

```
const string s = "Hello,my weibo name is hw-dong";
const auto it = s.find("Hello");
if (it != std::string::npos)
    std::cout << it << " Hello\n";

const auto it2 = s.find("hw-dong");
if (it2 != std::string::npos)
    std::cout << it2 << " hw-dong\n";
```

该代码中用不同名字 it 和 it2 区分 2 次查找的结果。C++17 中则可以写成下面的

形式：

```
const string s = "Hello,my weibo name is hw-dong";
if (const auto it = s.find("Hello"); it != std::string::npos)
    std::cout << it << " Hello\n";

if (const auto it = s.find("hw-dong"); it != std::string::npos)
    std::cout << it << " hw-dong\n";
```

上述代码在2个if语句中都使用了同样的表示位置的变量it，避免了代码中过多的变量名，每个it只属于它所在的if语句。其中string类型的find()函数用于在string对象s中查找一个字符串，string类型的变量npos即std::string::npos表示字符串的结束位置。

4.3 循环语句

4.3.1 while语句

1. while

用while关键字定义的while循环语句，其格式是：

```
while(表达式)
    程序块
```

当表达式的结果为true(或是能转换为true的非0值)时，就一直执行其中的程序块。例如，下面的程序可以计算一个输入整数n的阶乘：

```
//计算n的阶乘
//n的阶乘 n! = 1*2*3… *n
#include <iostream>
using namespace std;
int main() {
    int n, i{1}, factorial{1};
    cout << "请输入一个正整数: ";
    cin >> n;

    while (i <= n) {           //只要i小于或等于n,就一直执行while循环体
        factorial *= i;        //factorial = factorial * i;
        ++i;
    }

    cout << n <<"的阶乘是: = "<< factorial;
    return 0;
}
```

该程序初始化i值为1，然后只要i小于或等于n，就一直执行while循环体，不断将这个i乘到factorial上，并将i自增1。当i从1到n变化时，factorial的值就是1＊1＊2＊…＊n，即

n! 的值。

下面的程序表示从键盘中输入一组成绩,最后输出平均分:

```
auto score{0}, average{ 0 };
    auto num{ 0 };
    while (std::cin >> score) {
        average += score;
        num++;
}
std::cout << "平均成绩是: " << average / num << std::endl;
```

其中,只要 std::cin >> score 的结果即对象 cout 不处于异常或结束状态,while 里的语句会一直进行下去,不会结束。可以输入 Ctrl+Z (Windows 系统)或 Ctrl+D(UNIX 或 Mac OS 系统)使 cout 处于结束状态,终止循环。

2. break

前面看到,可以用 break 关键字跳出 switch 语句,同样,可以用 break 关键字跳出 while 循环语句。例如:

```
auto num{ 0 };
    while(std::cin >> score){
        if (score < 0)
            break;  //跳出 while 循环
        average += score;
        num++;
}
    std::cout << "平均成绩是: " << average/num << std::endl;
```

该程序从键盘中输入分数给 score 变量,但如果发现 score<0 则会跳出循环,即不再执行循环体。

3. do-while

while 语句的另外一个变种是所谓的 **do-while** 语句。

```
do
    程序块
while(表达式);
```

即先执行程序块,然后判断条件表达式是否为 true,如果为 true,则继续执行程序块,然后继续检查条件表达式是否为 true,决定是否继续执行语句(块),直到条件表达式的值为 false 或 0 时,才结束该语句的执行。

注意:*while(表达式)后面必须有分号";"。*

上面求平均分的程序可以写成等价的 do-while 语句形式:

```
double score, average{0};
auto num{0};
```

```
std::cin >> score;              //注意：先输入一个分数
do {
      if (score < 0)
            break;              //跳出 while 循环
      average += score;
      num++;
} while (std::cin >> score);
std::cout << "平均成绩是：" << average / num << std::endl;
```

当键盘输入的不是一个实数或输入的实数小于 0 时，都会使循环终止。

4. continue

关键字 **continue** 用于中断循环体里的后续语句执行，回到循环开头重新执行循环。例如下面的统计平均分的循环体，当输入负的分数时并不退出循环，而是继续回到循环开头，继续接受下一个输入，直到按 Ctrl+Z（Windows 系统）或 Ctrl+D(UNIX 或 Mac 系统)才停止执行。

```
int main() {
    double score, average = {0};
    auto num{0};
    while (std::cin >> score) {
        if (score < 0)
            continue;           //回到循环开头
        average += score;
        num++;
    }
    std::cout << "平均成绩是：" << average / num << std::endl;
}
```

4.3.2 for 语句

用 for 关键字也可以表示一个循环语句，其格式为：

```
for(初始化表达式；条件表达式；后处理表达式)
    程序块
```

其循环过程是：先执行初始化表达式，然后不断重复下列循环：如果条件表达式为 true（非 0 值），则执行循环体的程序块，然后执行后处理表达式，直到条件表达式为 false(0 值)，for 语句才执行结束。

例如下面用于计算 1～100 整数之和的代码：

```
auto s{0} ;
for(auto i{1} ; i <= 100; i++)
    s += i;
std::cout <<"1～100 的整数之和是："<< s << std::endl;
```

i 的初始值是 1，只要 i 的值小于或等于 100，就一直执行循环体，即将 i 加到变量 s 中，

并让 i 自增 1。当 i 大于 100 时才结束这个循环。

for 语句和 while 语句实质是等价的,上述格式的 for 语句可以转换为 while 语句:

```
初始化表达式;
while(表达式){
    程序块
    后处理表达式;
}
```

求平均分的程序可以写成等价的 for 语句形式:

```
double score,average{0};
auto num{0};
for(;std::cin >> score;){
    if(score < 0)
        break;              //跳出 while 循环
    average += score;
    num++;
}
std::cout <<"平均成绩是: "<< avarage/num << std::endl;
```

当然,上述 for 语句还可以改写成:

```
for(;std::cin >> score&&score >= 0;){
    average += score;
    num++;
}
```

可以看到,for 语句除中间的条件表达式外,“初始化表达式”和“后处理表达式”都可以省略。break 和 continue 关键字对于 for 循环和 while 循环的作用也是一样的。

例如,下面代码可输出 1~100 的所有被 3 整除的整数。

```
for(auto i{1}; i <= 100;i++){
    if(i % 3!= 0)
        continue;           //停止后续语句执行,回到循环的条件表达式"i <= 100"
    std::cout << i <<'std::endl';
}
```

4.4 跳转语句

有一个极少使用的用 goto 关键字定义的可以跳转到任何位置的 **goto 语句**,其格式是:

```
goto 标签名;
//...
标签名:
```

即当遇到“goto 标签名;”时,将跳到“标签名”处继续执行该标签后的语句。标签名由一个标识符和冒号(:)构成。

标签名也可以在 goto 语句的前面:

```
标签名:
//...
goto 标签名;
//...
```

如,可以用 goto 语句跳出一个循环体。

```
double score,average{0};
auto num{0};
for(;std::cin >> score;){
    if(score < 0)
        goto label;           //跳到标签 label 处执行
    average += score;
    num++;
}
label:
std::cout << "平均成绩是: " << average /num << std::endl;
```

除上面的这些控制语句,还有区间 for(range for)循环语句(见 5.3.5 节)、异常处理(try-catch)语句(见第 14 章)等将在后续章节介绍。

4.5 实战:控制台游戏——Pong 游戏

4.5.1 Pong 游戏

1972 年,美国的雅达利(Atari)公司开发了一个模拟两个人打乒乓球的街机游戏 Pong,该游戏被认为是游戏机历史的起点。该游戏就是在两条线中间有一个运动的代表球的点,玩家通过摇杆的按钮操纵两边的挡板将球回击到对方区域。最初,Atari 的两名老板骗一个刚招聘来的小青年 Alcorn 说是为 GE for Atari 制作游戏,没有任何游戏开发经验的 Alcorn 二话没说,立即全身心地投入制作之中,两名老板对于 Alcorn 的用心非常欣赏,虽然他们并不十分有信心,但也制作了一个 Pong 样板机,放在酒吧进行测试,当第一台 Pong 游戏机被安装在酒吧后,非常火爆,导致酒吧人满为患。Atari 的老板看到了巨大的商机,经过四处筹款,开始扩大生产线。很快,一台台 Pong 出现在了美国的各个角落,在美国掀起了一场风暴。Pong 把电子游戏的观念第一次带给了普通民众,引来了其他厂商的纷纷效仿,远在彼岸的日本,也感受到了 Pong 的吸引力,娱乐公司 Taito 开发了一个类似的产品 Elepong,成为日本的第一款电子游戏。

图 4-1 所示是要完成的 Pong 游戏的画面,其中有 1 个乒乓球(ball)和左右两个挡板(paddle),乒乓球从区域中心开始随机运动,左右挡板分别由两个游戏者操作,试图将球击向对方,如果能成功击回,就得一分,如果未能挡住,将失去一分。当球跑出画面后,新的球

又从一个随机位置再一次沿随机方向运动。

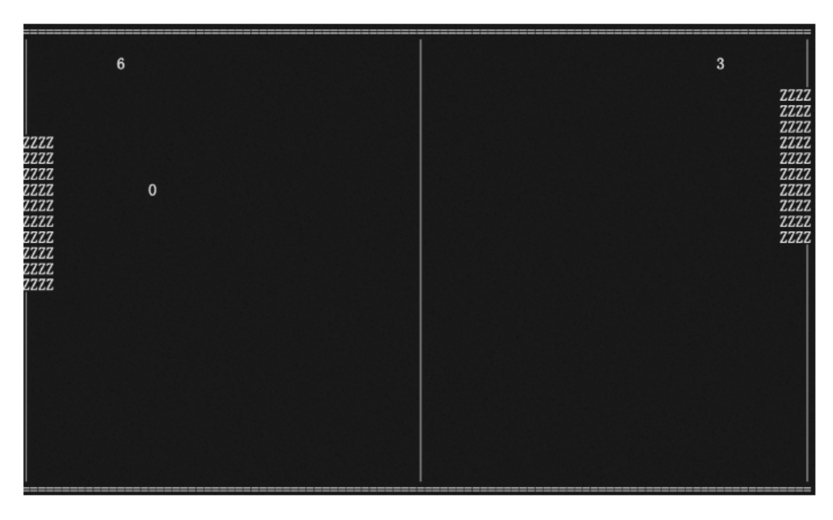


图 4-1 Pong 游戏画面

4.5.2 初始化

游戏画面有一个窗口，窗口里除了一些背景(如 Pong 游戏中的分隔线)外，主要运动物体是一只球(ball)和 2 个挡板(paddle)，分别用 1 个圆和 2 个矩形表示。球有位置、大小(半径)、颜色、速度等属性，而挡板也有位置、大小(长宽)、颜色、速度等属性。游戏还有记录各自分数的变量。当然，游戏窗口也有长宽、标题、背景颜色等属性。游戏开始时，需要对这些数据初始化。

```
#include <iostream>
using namespace std;
int main() {
    //1. 初始化游戏中的数据
    auto WIDTH{ 120 }, HEIGHT{ 40 };                                        //窗口长宽
    auto ball_x {WIDTH/2}, ball_y{HEIGHT/2}, ball_vec_x{0}, ball_vec_y{0}; //球位置及速度
    auto paddle_w{4}, paddle_h{10};                                         //挡板的长宽
    auto paddle1_x{0}, paddle1_y{HEIGHT/2 - paddle_h/2}, paddle1_vec{3};    //挡板 1 位置及速度
    auto paddle2_x{ WIDTH - paddle_w },
              paddle2_y{ HEIGHT/2 - paddle_h/2 }, paddle2_vec{3};           //挡板 2 位置及速度
    auto score1{ 0 }, score2{ 0 };                                          //双方的得分
    return 0;
}
```

4.5.3 绘制场景

绘制场景包括绘制背景和游戏中的运动物体。可以先绘制背景中的上下墙壁和 3 条竖线。

```
int main() {
    //…
```

```
    //2. 绘制场景
    //2.1 绘制背景
    //2.1.1 绘制背景中的顶部墙
    for (auto x = 0; x <= WIDTH; x++)
        std::cout << '=';
    std::cout << '\n';
    //2.1.2 绘制背景中的 3 条竖线
    for (auto y = 0; y <= HEIGHT; y++) {
        for (auto x = 0; x <= WIDTH; x++)
            if (x == 0 || x == WIDTH / 2 || x == WIDTH)
                std::cout << '|';
            else std::cout << ' ';
        std::cout << '\n';
    }
    //2.1.3 绘制背景中的底部墙
    for (auto x = 0; x <= WIDTH; x++)
        std::cout << '=';
    std::cout << '\n';
}
```

也就是通过在控制台窗口中输出一些特殊字符，如|或=，分别表示 3 条竖线和墙的图案。图 4-2 显示的是背景画面。

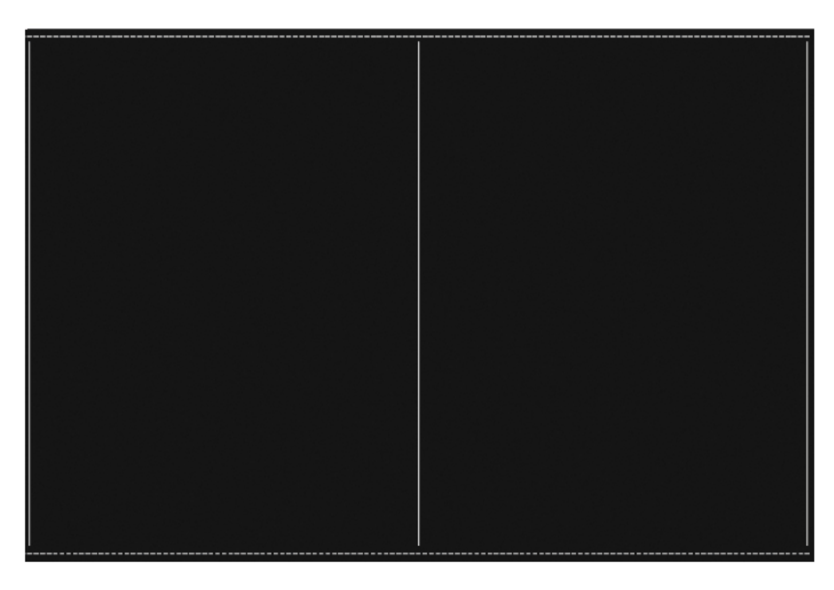


图 4-2 背景画面

挡板可以用一个矩形表示，球可以用一个圆表示，如何绘制圆形的球？一种简单的方法是用一个字母如大写的 O 表示一个球，当然也可以绘制一个更大的圆的图案。

为了将球和挡板绘制在窗口画面中，需要在前面的绘制背景的循环中判断哪些位置是球和挡板，然后在这些位置绘制代表球和挡板的特殊图案字符。因此，需要修改前面的 if-else 语句，在球和挡板的位置绘制球和挡板。

```
for (auto y = 0; y <= HEIGHT; y++) {
    for (auto x = 0; x <= WIDTH; x++)
```

```
        if(x == ball_x&&y == ball_y)                        //球的位置
            std::cout << 'O';
        else if (y >= paddle1_y &&y < paddle1_y + paddle_h
            &&x >= paddle1_x && x < paddle1_x + paddle_w) {  //左挡板位置
            std::cout << 'Z';
        }
        else if (y >= paddle2_y && y < paddle2_y + paddle_h
            && x >= paddle2_x && x < paddle2_x + paddle_w) { //右挡板位置
            std::cout << 'Z';
        }
        else if (x == 0 || x == WIDTH  / 2 || x == WIDTH)    //竖线位置
            std::cout << '|';
        else std::cout << ' ';
    std::cout << '\n';
}
```

现在画面上出现了球和挡板(见图 4-3)。

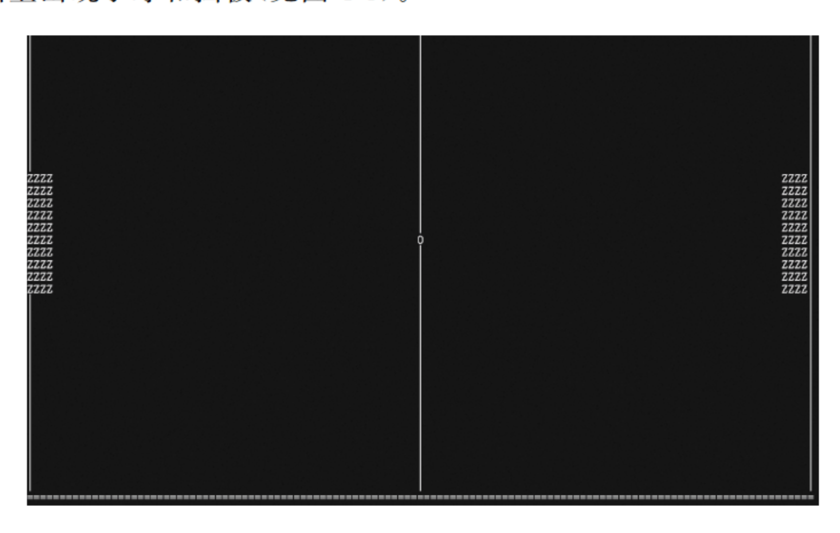


图 4-3 球和挡板

4.5.4 让球动起来

上面的程序中球和挡板是静止的,如何让它们动起来?游戏中是通过不断绘制新的画面让游戏画面动起来的。游戏实际是如下的一个循环过程:

```
初始化游戏数据
循环(直到游戏结束){
    处理事件(如用户输入、定时器)
    更新游戏状态(游戏中的数据)
    绘制游戏画面
}
```

为了绘制新的画面,必须先清除原来的画面。在 Windows 平台上,可以用如下 gotoxy() 函数(需要包含 windows.h 头文件)。

```
#include <windows.h>
void gotoxy(int x, int y){
    COORD coord = {x, y};
    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), coord);
}
```

只要在每次绘制新的画面前，调用这个函数 gotoxy(0,0)将光标定位在左上角就相当于清除了屏幕的内容。

另外，为了防止画面刷新时出现闪烁光标，可以使用下面的函数隐藏掉光标：

```
void hideCursor(){
    CONSOLE_CURSOR_INFO cursor_info = {1, 0};
    SetConsoleCursorInfo(GetStdHandle(STD_OUTPUT_HANDLE), &cursor_info);
}
```

为了给球一个随机的初始速度，需要用到随机数生成的相关函数，生成随机的初始化速度。

```
srand((unsigned)time(0));              //生成随机数种子
ball_vec_x = rand() % 3 + 1;           //生成一个随机整数，表示 x 和 y 方向的速度大小
ball_vec_y = rand() % 3 + 1;
if (rand() % 2 == 1) ball_vec_x = -ball_vec_x;    //随机改变初始的速度方向
if (rand() % 2 == 1) ball_vec_y = -ball_vec_y;
```

其中的 srand()函数用于生成一个随机数种子，然后用 rand()函数生成一个整数，通过%运算，使得代表速度的整数不至于过大。

在游戏循环中根据速度不断更新球的位置，并绘制游戏画面，就能让球动起来。

```
ball_x += ball_vec_x;                  //根据速度改变位置
ball_y += ball_vec_y;
```

完整代码如下：

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <windows.h>
using namespace std;

void gotoxy(int x, int y) {
    COORD coord = { x, y };
    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), coord);
}
void hideCursor() {
    CONSOLE_CURSOR_INFO cursor_info = { 1, 0 };
    SetConsoleCursorInfo(GetStdHandle(STD_OUTPUT_HANDLE), &cursor_info);
```

```
}

int main() {
    //1. 初始化游戏中的数据
    auto WIDTH{ 120 }, HEIGHT{ 40 };       //窗口长宽
    auto ball_x = {WIDTH/2}, ball_y{HEIGHT/2}, ball_vec_x{}, ball_vec_y{};
    auto paddle_w{4}, paddle_h{10};
    auto paddle1_x{0}, paddle1_y{HEIGHT/2 - paddle_h/2}, paddle1_vec{3};
    auto paddle2_x{WIDTH - paddle_w},
              paddle2_y{HEIGHT/2 - paddle_h/2},paddle2_vec{3};

    srand((unsigned)time(0));             //生成随机数种子
    ball_vec_x = rand() % 3 + 1;          //生成一个随机整数
    ball_vec_y = rand() % 3 + 1;
    if (rand() % 2 == 1) ball_vec_x = - ball_vec_x;
    if (rand() % 2 == 1) ball_vec_y = - ball_vec_y;

    //游戏循环
    while (true) {
        //1. 处理事件

        //2. 更新数据
        ball_x += ball_vec_x;
        ball_y += ball_vec_y;

        gotoxy(0, 0);                     //定位到(0,0),相当于清空屏幕
        hideCursor();                     //隐藏光标
        //3. 绘制场景
        //3.1 绘制背景
        //3.1.1 绘制背景中的顶部墙
        for(auto x = 0; x <= WIDTH; x++)
            std::cout << '=';
        std::cout << '\n';
        //3.1.2 绘制背景中的 3 条竖线、球、挡板
        for (auto y = 0; y <= HEIGHT; y++) {
            for (auto x = 0; x <= WIDTH; x++) {
                if (ball_x == x && ball_y == y)
                    std::cout << 'O';
                else if (y >= paddle1_y && y < paddle1_y + paddle_h
                    && x >= paddle1_x && x < paddle1_x + paddle_w) {
                    std::cout << 'Z';
                }
                else if (y >= paddle2_y && y < paddle2_y + paddle_h
                    && x >= paddle2_x && x < paddle2_x + paddle_w) {
                    std::cout << 'Z';
                }
                else if (x == 0 || x == WIDTH / 2 || x == WIDTH)
                    std::cout << '|';
```

```
                    else std::cout << ' ';
                }
                std::cout << '\n';
            }

            //3.1.3 绘制背景中的底部墙
            for (auto x = 0; x <= WIDTH; x++)
                std::cout << '=';
            std::cout << '\n';
        }
        return 0;
    }
```

当运行程序时，球会跑出画面而不再可见。而在 Pong 游戏中，球如果和上下墙壁发生碰撞会反弹回来，而遇到挡板也会发生反弹，如果没有遇到挡板，则球跑出画面，另外一方的得分将增加，然后球重新从新位置以随机速度出发。

先检测球是否和墙壁或挡板碰撞，如发生碰撞，则通过改变球的相应的速度方向，而使球发生反弹。和上下墙壁碰撞，球的垂直速度方向变成反方向；和左右挡板碰撞，球的水平速度方向变成反方向。

```
    //2. 更新数据
    ball_x += ball_vec_x;
    ball_y += ball_vec_y;
    if (ball_y < 0 || ball_y >= HEIGHT)        //和上下墙碰撞,改变垂直速度方向
          ball_vec_y = -ball_vec_y;

    if (ball_x < paddle_w&& ball_y >= paddle1_y && ball_y < paddle1_y + paddle_h)
    {//和左挡板碰撞,改变水平速度方向
          ball_vec_x = -ball_vec_x;
          score1 += 1;
    }
    else if(ball_x > WIDTH - paddle_w && ball_y >= paddle2_y
                && ball_y < paddle2_y + paddle_h)
    { //和右挡板碰撞,改变水平速度方向
          ball_vec_x = -ball_vec_x;
          score2 += 1;
    }
      bool is_out{ false };                     //是否跑出沟渠的bool标志
      if (ball_x < 0) {score2 += 1; is_out = true;}
      else if (ball_x > WIDTH - paddle_w) {score1 += 1; is_out = true;}
      if (is_out) {                             //跑出左右沟渠,球回到中心并以新的随机速度出发
            ball_x = WIDTH / 2; ball_y = HEIGHT / 2;
            ball_vec_x = rand() % 3 + 1;
            ball_vec_y = rand() % 3 + 1;
            if (rand() % 2 == 1) ball_vec_x = -ball_vec_x;
            if (rand() % 2 == 1) ball_vec_y = -ball_vec_y;
    }
```

运行该程序，就可以看到小球在窗口里运动了（见图 4-4）。

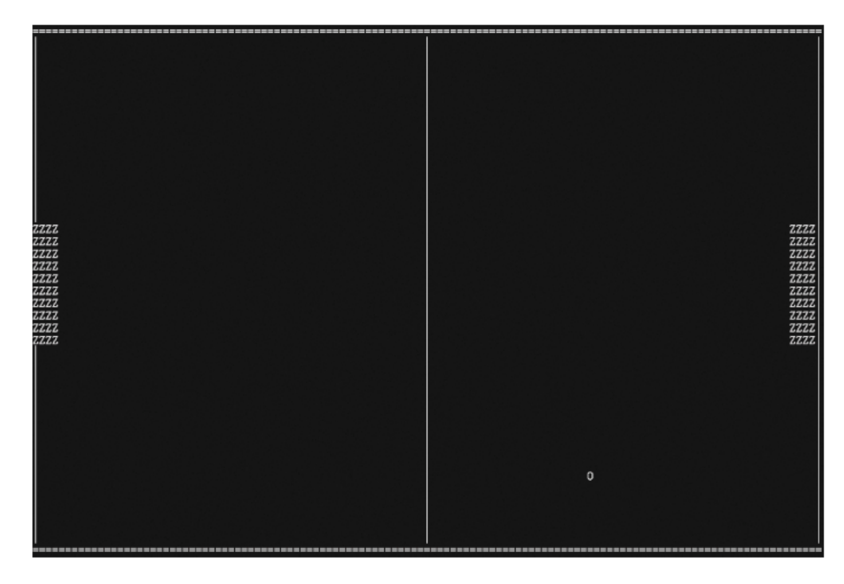


图 4-4　运动的球

4.5.5　事件处理：用挡板击打球

为了让挡板运动，就需要处理事情，如让用户通过键盘输入控制挡板的运动。假如用上、下箭头键移动右挡板，而用字母 w 和 s 移动左挡板。为了得到键盘输入，先用 kbhit()函数检测是否存在按键消息，然后通过 getch()函数得到按键的字符。这两个函数在头文件 conio. h 中说明。

下面代码根据用户的输入改变挡板的位置：

```
/ 1. 处理事件
char key;
if (_kbhit()) {                          //键盘有输入
      key = _getch();                    //得到输入的键值
      if ((key == 'w'|| key == 'W')&& paddle1_y > paddle1_vec)
            paddle1_y -= paddle1_vec;
      else if ((key == 's' || key == 'S')&& paddle1_y + paddle1_vec + paddle_h
            < HEIGHT)
            paddle1_y += paddle1_vec;
      else if (key == 72&& paddle2_y > paddle2_vec)
            paddle2_y -= paddle2_vec;
      else if ((key == 80)&& paddle2_y + paddle2_vec + paddle_h < HEIGHT)
            paddle2_y += paddle2_vec;
}
```

如何显示分数？首先定义分数以及显示分数位置的变量：

```
auto score1{ 0 }, score2{ 0 }, score1_x{ paddle_w + 8 }, score1_y{ 2 },
                             score2_x{ WIDTH - 8 - paddle_w }, score2_y{ 2 };
```

要将 int 型的分数转换为字符串才能通过输出字符串的方式显示分数，可以使用 C++标准库的字符串类型 string 的 to_string()函数将一个整数转换为一个字符串。

```
std::string s1{ std::to_string(score1) }, s2{ std::to_string(score2) } ;
```

最后在绘制场景时，在显示分数的位置输出这个表示分数的字符串即可：

```
else if (y == score1_y && x == score1_x) {        //左分数位置
     std::cout << s1;
     while (x < score1_x + s1.size()) x++;
     x--;
}
else if (y == score2_y && x == score2_x) {        //右分数位置
     std::cout << s1;
     while (x < score2_x + s2.size()) x++;
     x--;
}
```

完整代码请在作者的网站上下载。

4.6 习题

1. 什么是空语句？什么时候需要用到空语句？

2. 什么是程序块？为什么有时要定义程序块？

3. 编写一个简单的计算器程序，依次输入左操作数、运算符、右操作数，输出程序运算结果。

4. 编写程序，从键盘输入一个年份，根据年份是否是闰年，分别打印不同的信息。

5. 编写程序，从键盘输入一个人的身高和体重，根据 BMI 指数公式，输出这个人是否肥胖的判断结果。

注：胖瘦的 BMI 指数＝体重(kg)/身高^2(m)。BMI 指数衡量胖瘦的 WTO 标准：偏瘦(BMI<18.5)、正常(18.5≤BMI< 25)、偏胖(25≤BMI<30)、肥胖(BMI>30)。

6. 编写程序，从键盘输入摄氏温度，将其转换为华氏温度并输出。

7. 输入一行字符，分别统计出其中英文字母、空格、数字和其他字符的个数。

注：C++判断一个字符是否字母、数字、空格的函数分别为 isalpha()、isdigit()、isspace()，返回的是 true 和 false。头文件 cctype 中声明了这些函数。输入一个字符，也可以用 if(ch >='a'&&ch <='z')来判断一个字符是否是小写字母。

8. 下列代码片段的错误是什么？为什么？

```
case true:
    int ival{0};
    int jval;
    break;
```

```
case false:
    jval = 3;
```

9. 请分别写出用 for 语句和 do-while 语句计算 n 的阶乘的程序。

10. 下列的代码片段的错误原因是什么？请改正。

(1) `while (int i == 3) {/ * … * /}`

(2) `while (int j = 3) {/ * … * / }`
```
        j++;
```

(3) `for (unsigned i{ 10 }; i >= 0; i-- )`
```
            std::cout << i <<'\n';
```

11. 说明下列程序的错误及其原因。

```
#include <iostream>
int main() {
    int arr[10];
    for (int i{ 0 }; i < 10; i++)
        arr[i] = 2 * i + 1;
    for (i = 0; i < 10; i++)
        std::cout << i << '\t';
}
```

12. 如下所示，编写程序根据用户输入行数，输出对应行数的金字塔。

```
      *
    *   *
  *   *   *
*   *   *   *
```

13. 下面是 6 行的杨辉三角形，编写程序，输入行数打印对应行数的杨辉三角形。

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5   10  10  5   1
```

14. 编写一个程序，从键盘输入一个正整数，判断它是否是质数。

提示：假设 n 不是质数，必然存在 $1<a<b<n$，使得 $n=a\times b$。

15. 编写一个程序，输出小于 100 的所有质数。

16. 编写程序：根据公式 PI/4=1−1/3+1/5−1/7…，求出圆周率 PI 的近似值。要求最后一项的绝对值小于 10^{-6}，输出 PI 的近似值。

17. 猜数字游戏：下列程序随机生成一个 1～100 的正整数 num，然后让用户从键盘输入一个猜想的数字 guess，如果 guess 等于 num，那么就显示成功的祝贺信息，如果失败就提示用户继续输入，直到超过指定的猜测次数（如 8 次）就提示失败的信息。请在？处补充代码。

```
#include <iostream>
#include <cstdlib>
#include <ctime>

int main() {
    srand((unsigned)time(0));                    //生成随机数种子
      int number = rand() % 100 + 1;             //生成一个[1,100]的随机整数
      int guess{};
      bool success = false;                      //是否成功的标志
      for (int i = 0; i != 8 ?; i++) {
          std::cout << "请输入猜测的数字：";
          std::cin >> guess;
          ?                                      //判断是否猜中
      }
      if (?)
          std::cout << "祝贺,你猜中了这个数：" << number << '\n';
      else
          std::cout << "很遗憾,猜测失败!";
}
```

执行情况：

```
请输入猜测的数字：50
你猜测的数字有点小：
请输入猜测的数字：70
你猜测的数字有点大：
请输入猜测的数字：60
你猜测的数字有点大：
请输入猜测的数字：55
你猜测的数字有点大：
请输入猜测的数字：52
你猜测的数字有点小：
请输入猜测的数字：53
你猜测的数字有点小：
请输入猜测的数字：54
祝贺,你猜中了这个数:54
```

18. 从键盘输入一个弧度 x，计算正弦函数 $\sin(x)$的值。要求最后一项的绝对值小于 10^{-5}，并统计出此时累计了多少项。$\sin(x)$的近似计算公式为：

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \cdots$$

19. 从键盘输入一个正整数 n，计算从 1 到 n 的所有整数的阶乘之和。

20. 有 5 个人坐在一起，问第 5 个人多少岁，他说比第 4 个人大 2 岁。问第 4 个人多少岁，他说比第 3 个人大 2 岁。问第 3 个人，他说他比第 2 人大 2 岁。问第 2 个人，他说比第 1 个人大 2 岁。最后问第 1 个人，他说是 10 岁。请问第 5 个人多少岁？

21. 约瑟夫环问题：有 n 个人围成一圈，按顺序编号。从第 1 个人开始报数（从 $1\sim m$ 报数），凡报到 m 的人退出圈子，问最后留下的是原来的第几号？

22. 海滩上有一堆桃子，5 只猴子来分。第 1 只猴子把这堆桃子平均分为 5 份，多了 1 个，这只猴子把多的 1 个扔入海中，拿走了 1 份。第 2 只猴子把剩下的桃子又平均分成 5 份，又多了 1 个，它同样把多的 1 个扔入海中，拿走了 1 份，第 3～5 只猴子都是这样做的。试问，海滩上原来最少有多少个桃子？



复合类型：数组、指针和引用

C++的内在类型还包含从基本类型派生出来的复合类型：数组、指针和引用。

5.1 引用

引用(reference)就是一个变量(对象)的别名。假如有一个数据类型 T 的变量 var，下面的语句：

```
T& ref{var};
```

定义了一个**引用变量** ref，它是变量 var 的一个别名。在定义变量 ref 时，其前面的符号 & 表示 ref 是一个引用变量(即其他变量的别名)。引用变量 ref 的数据类型是 T& 而不是 T。

当然，定义引用变量时可以用不同的初始化方式，如上述定义也可以写成：

```
T& ref = var;
```

再看下面的例子：

```
int ival{1024};
int &ref{ival};         //int 类型的引用变量 ref 是变量 ival 的别名
```

既然引用变量引用的是其他变量(即引用变量是其他变量的别名)，那么定义引用变量时就必须指定它引用的是哪一个变量，不指定引用的变量是错误的。如：

```
int &ref2;              //错: int 类型的引用变量 ref2 没有初始化
```

对引用变量的操作就是对它引用的那个对象的操作。如：

```
int ival{1024};
int &ref{ival};         //ref 引用 ival,是 ival 的别名
```

```
ref = 24;              //也就是 ival = 24,因为 ref 和 ival 是同一块内存的不同名字而已
int ii{ref};           //相当于 ii{ival}
int &ref3{ref};        //相当于 int& ref3 = ival,即 ref3 和 ref 一样都是 ival 的别名
```

一个语句里可以定义多个引用。如：

```
int i{1024},i2{2048};
int &r{i},r2{i2};      //r 引用 i,r2 是普通 int 类型变量,不是引用
int i3{24},&ri{i3};    //i3 不是引用,ri 引用 i3
int &r3{i3},&r4{i2};   //r3 引用 i3,r4 引用 i2
```

引用变量必须引用一个变量(对象),而不能是文字量。另外,引用变量类型和被引用变量类型应该一致。如：

```
int &ref4{10};         //错：不能引用文字量
double dval{3.14};
int &ref5{dval};       //错：引用变量类型和被引用变量类型不一致
```

引用变量一旦定义,就不能再引用其他变量,即不能"重定义"。如：

```
int a,b;
int &ra{a};
int &ra{b};            //错：不能重定义同一个引用变量 ra
```

5.2 指针

5.2.1 指针类型

对于一个类型 T,T * 是 **T 指针**类型,即 T * 类型的变量可以保存 T 类型变量的地址。

```
char c = 'a';
char *p = &c;          //p 的类型是 char * ,它初始化为 char 类型变量 c 的地址(指针)
                       //其中 & 是取地址运算符,用于获得一个变量(如 c)的地址(指针)
```

& 是取地址运算符,它作用于一个变量,可以得到这个**变量的地址**,因此,表达式 &c 的结果是变量 c 的地址。变量 p 的初始值就是变量 c 的**地址**(地址也可称为**指针**),习惯上称为"p 指向 c",而变量 p 习惯上称为**指针变量**,其含义是该变量保存的是其他变量的指针(地址),如图 5-1 所示。

对于一个 T 类型的变量 var,取地址运算符 & 作用于它(&var)的结果值的类型是 T * ,习惯上称数据类型 **T *** 为 **T 指针类型**,而 **T *** 类型的变量就是 **T 指针类型**的变量。

图 5-1　指针变量 p 存储变量 c 的地址

注意：T * 和 T 是完全不同的两个类型,相互之间不能初始化或赋值。因此,下面的语

句都是错误的：

```
char *q {c};            //不能用 char 类型的值初始化 char * 类型的变量
p = c;                  //也不能将 char 类型的值赋值给 char * 类型的变量
char ch{p};             //char 类型变量也不能用 char * 类型值初始化
ch = p;                 //也不能用 char * 类型值赋值给 char 变量
```

一句话，T 和 T * 是 2 个完全不同的类型，这 2 个不同类型的变量之间不能相互初始化或赋值。

p 中保存了变量 c 的地址，可以通过**解引用运算符** * 作用于指针变量 p，得到它指向的那个变量，即 ***p** 就是变量 c。如：

```
char c{'a'};
char *p{&c};            //p 存储的是 c 的地址，即 p 指向 c
*p = 'A';               //*p 就是 c，因此，相当于 c = 'A'。即变量 c 的内存块存储的内容是字符'A'
char c2{*p};            //相当于 char c2 {c}. 即 c2 的初始值就是 c 的值，也即字符'A'
```

和普通变量一样，指针变量也占据一块独立的内存。因此，定义指针变量时，不一定要初始化。而引用变量仅仅是其他变量的别名，引用变量本身不占据单独的一块内存，引用变量定义时则必须初始化。如：

```
double *s;              //指针变量定义时可以不初始化
double &r;              //错：引用变量定义时必须指明引用哪个变量
```

给指针变量初始化和赋值时，类型须相同或能隐含转换。如：

```
double d;
double *pd;
pd = &d;                //OK:pd 的类型是 double *，&d 的类型是 double *。类型完全相同
double *pd2 = pd;
int *pi = pd;           //错：pi 的类型是 int *，而 pd 的类型是 double *。类型不一致
pi = &d;                //错：pi 的类型是 int *，而 &d 的类型是 double *。类型不一致
```

注意：* 和 & 的多个含义。

- 定义变量时，* 和 & 分别表示定义指针变量和引用变量。
- 作为运算符时，* 和 & 则分别表示解引用和取地址运算符。

```
int i{56};
int &r{i};              //r 引用 i，即 r 是变量 i 的别名，r 和 i 是同一块内存的不同名字
int *p;                 //p 的类型是 int *，可存储 int 型变量的地址

p = &i ;                //&i 得到 int 类型变量 i 的地址，赋值给变量 p。两者类型都是 int *
*p = 3;                 //*p 得到 p 指向的那个变量，即 i。因此这句命令相当于 i = 3;
int  &r2{*p};           //int 类型引用变量 r2 引用的变量是 *p，即 i
```

空指针：不指向任何变量（对象）的指针（变量）。初始化一个空指针的方法很多，示例

如下：

```
int *p2{0};             //用 0 初始化一个空指针变量
int *p1{nullptr};       //从 C++11 开始,用 nullptr 表示空指针。推荐使用
int *p3{NULL};          //NULL 通常是一个为 0 的宏常量。C++11 开始禁止这样使用
```

从 C++11 开始，推荐用专门的 nullptr 表示空指针，可以用它初始化或赋值给任何指针类型变量，但不能赋值给普通变量。

```
int *pi{nullptr};
double *pd{nullptr};
int ival{nullptr};      //错: ival 不是一个指针变量
```

不能用整数给指针赋值，即使这个整数为 0，因为类型不同。如：

```
int zero{0} , *p1;
p1{zero};               //错: p1 类型是 int * ,而 zero 类型是 int
int *p2{2};             //错: p2 类型是 int * ,而 2 的类型是 int
```

上述代码用 int 类型变量初始化或给 int * 变量赋值都是错误的。

5.2.2 指针的其他运算

和非 0 值一样，非空指针可以自动转换为 bool 类型的值 true；和 0 一样，空指针可以自动转换为 bool 类型的值 false。如：

```
int i;
int *p{&i}, *q{0};
bool b{p};              //int * 非空指针 p 转换为 bool 型值 true
                        //然后对 b 初始化,因此,b 的值是 true
std::cout << boolalpha << b << std::endl;    //boolalpha 操作符控制 bool 量的显示形式
b = q;                  //int * 空指针 q 转换为 bool 型值 false
                        //然后赋值给 b,因此,b 的值是 false
std::cout << boolalpha << b << std::endl;
```

指针类型的变量可以用比较运算符(!=、==、>=、<等)进行比较。结果是一个逻辑值 true 或 false。如：

```
std::cout << boolalpha << (p!= q) << std::endl;
```

指针可以和整数进行加减运算，用于对指针进行偏移(在数组和动态内存分配时会再进一步介绍)。

5.2.3 void * 无类型指针

- void * 变量可以存储任何内存地址(任何类型对象的地址)。
- void * 变量之间可以比较大小，可以相互赋值。

- void * 变量不能解引用、不能对指针进行偏移、不能隐式类型转换到非 void 指针类型，但可以通过强制类型转换 **static_cast** 将它转换到特定的指针类型。

```
int main(){
    int *pi;
    void *pv = pi;         //OK: int * 到 void * 的隐式类型转换
    * pv;                  //错:不能解引用 void *
    ++pv;                  //错:不能增量或偏移 void * (指向对象的内存大小未知)
    int * pi2 = static_cast<int *>(pv);  //void * 强制类型转换到 int *
    double * pd1 = pv;     //错:不能将 void * 初始化或赋值给非 void * 指针变量
    double * pd2 = pi;     //错: 指针类型不一致
    double * pd3 = static_cast<double *>(pv);   //不安全
}
```

void * 指针变量主要用于将不同类型的指针变量传递给函数，在函数内部再将它强制转换为特定的指针类型。在后续函数章节再介绍这种用法。

5.2.4 指针的指针

既然指针变量 pi 也是占据独立内存块的变量，它本身的地址 &pi 也可以保存在一个指针变量 ppi 中，这个指针变量 ppi 通常称为**“指针的指针”**，也就是说 ppi 存储的是一个“指针变量的地址”。如：

```
#include <iostream>
using namespace std;
int main(){
    int ival{1024};
    int *pi{&ival};        //pi 存储 ival 的地址
    int **ppi{&pi};        //ppi 存储 pi 的地址。pi 的类型是 int *
                           //所以 &pi 的类型是(int * ) * ,即 int ** ,int ** 就是(int * ) *
                           //ppi ---> pi --> ival
    cout <<"ival 的值是: "<< ival << endl;
    cout <<"ival 的值是: "<< * pi << endl;     //*pi 就是 ival
    cout <<"ival 的值是: "<< ** ppi << endl;   //**ppi 即 * ( *ppi),而 *ppi 就是 pi,因此 **ppi
                                               //就是 *(pi),即 ival

    cout <<"\nival 的地址是: "<< &ival << endl;
    cout <<"ival 的地址是: "<< pi << endl;          //pi 保存的是 ival 的地址
    cout <<"ival 的地址是: "<< * ppi << endl;       //*ppi 就是 pi

    cout <<"\npi 的地址是: "<< &pi << endl;
    cout <<"pi 的地址是: "<< ppi << endl;   //ppi 保存的是 pi 的地址
}
```

其中，指针变量 ppi 的类型是 int ** ，即 int * 类型的指针类型，相当于(int *) * 。因此，它可以存放 int * 类型变量的地址，而 pi 的类型正好是 int * ，因此，可以将 pi 变量的地址存储在这个变量 ppi 中，即 int **ppi{&pi}。

5.2.5　指针的引用

指针既然是一个占有独立内存的变量(对象)，当然可以定义一个引用它的引用变量，即给它起一个引用别名。如：

```
int i{42};
int *p;
int *&r{p};          //r引用p,即r是p的别名
r = &i;              //将i的地址赋值给r,也就是p,因此p指针变量里保存的就是i的地址
 *r = 0 ;            //相当于*p = 0,p指针变量的值是0,即p成为一个空指针
```

理解变量r的类型的方式是"**从右向左看**"。紧靠r的是符号&，说明r首先是一个引用变量，再往左看是int *，说明r引用的是int *类型变量。注意，下面用法是错误的。

```
int &*q;               //错：因为从右向左看,q是一个指针变量,存储的是int &变量的地址
    //也就是说q试图存储一个引用变量的地址,而引用变量是没有独立的内存块的
    //即引用变量没有地址
```

5.2.6　引用和指针的比较

- 共同点：都是间接指向或引用其他对象。
- 不同点：引用(变量)仅仅是其他变量的别名，无独立内存，一个引用变量不能被修改去引用不同的变量。指针变量存储其他变量的地址，有独立内存，在不同时刻可指向不同对象。

理解并编译下面的程序，看看有哪些编译错误。

```
int main(){
    auto i{0},j{1};
    int *p;              //指针变量不一定要初始化
    int &r{i},&r1;       //错：引用变量r1没有初始化
    p = &i;              //p指向i
    p = &j               //p指向j
    auto * &rp{p};       //rp引用p
    int * &rp2;          //错：引用变量rp2没有初始化
    int &*q;             //错：不能定义指向引用的指针
                         //因为引用变量没有独立内存(即没有地址)
    int &*q2 = &r;       //错：原因同上。另外,取地址运算符&不能作用于引用变量r
}
```

5.3　数组

5.3.1　数组和下标运算符

对于一个类型T，T[size]是"size个T类型元素的数组"类型。T[size]类型的变量var

定义为 T var[size]而不是 T[size] var。即：

```
T var[size];                //var 是 T[size]类型的变量,即 "size 个 T 类型元素的数组"
```

例如：

```
float v[3];                 //v 是 3 个 float 类型元素的一个数组
char * a[32];               //a 是 32 个 char * 类型元素的一个数组,每个元素的类型是 char *
```

可以用**下标运算符 operator[]**访问数组的元素，下标从 0 开始。设 var 是“n 个 T 类型数据元素的数组”，则数组 var 有 n 个元素，其下标分别是 0、1、2、……、n－1。可以用 var[0]、var[1]、……、var[n－1]访问数组 var 的每个元素。超出下标范围的访问（如 var[－1]、var[n]等）都是非法的。如：

```
float v[3];                 //数组 v 的 3 个元素分别是: v[0],v[1],v[2]
char * a[32];               //数组 a 的 32 个元素分别是: a[0],a[1],…,a[31]
v[1] = 10;                  //v 的第二个元素的值修改为 10
std::cout << v[0]<'\t'<< v[1]<<'\t'<< v[2]<< std::endl;      //输出 v 的 3 个元素
std::cout << v[3];          //错: 下标超出范围
std::cout << v[ - 1];       //错: 下标超出范围
a[1] = 0 ;                  //a 的第 2 个元素成为空指针
a[2] = 'a';                 //错: 不能将 char 类型的值赋值给 char * 类型的元素
auto b = v[2];              //用 v[2]对变量 b 进行初始化,因此 b 是 float 类型的变量
```

数组的大小必须是“**常量表达式**”，即编译时值确定的表达式。如：

```
int s = 20;
int arr[s];                 //错: s 不是常量表达式(编译时常量)
int arr2[20];               //OK: 文字量 20 是编译时常量
```

所谓**编译时常量**，指编译时就能确定值的量且今后不可能被修改。而上述的 s 是普通的变量，在程序中是可能变化的，而文字量 20 是常量表达式（即编译时常量）。

和普通变量一样，在定义数组变量时，也可以对它初始化，即用花括号{ }括起来的**列表**对其每个元素进行初始化。此时，如果不指定数组的大小，其大小由列表中的元素个数决定。

```
int v1[]{1,2,3};                  //v1 是 3 个 int 类型元素的数组
char v2[]{'a','b','c','\0'};      //是 4 个 char 类型元素的数组,最后一个转义字符'\0'称为
                                  //结束字符,其 8 位二进制都是 0
char v3[2] {'a','b','\0'};        //错: 列表中的元素个数不能超出其大小
char v4[4] {'a','b','\0'};        //OK,4 个元素的数组
int v5[4]  {1,2,3};               //列表中的个数少于数组大小,剩余的数组元素的值取默认值
                                  //对于内在类型,默认值通常是 0,即等价于 int v5[4] = {1,2,3,0}
cout << v5[0]<<'\t'<< v5[1]<<'\t'<< v5[2]<<'\t'<< v5[3]<< endl;
```

注意：不能用一个数组去初始化或赋值给另一个数组。

```
int a[] = {1,2,3};
int a2[] = a;                //错：不能用一个数组去初始化另一个数组
a2 = a;                      //错：不能用一个数组去赋值给另一个数组
```

字符数组可以用一个字符串文字量进行初始化。

```
char a1[]{'C','+','+'};      //a1 是 3 个 char 字符的数组
char a2[]{'C','+','+','\0'}; //a2 是 4 个 char 字符的数组,最后一个字符是结束字符
char a3[]{"C++"};            //用字符串文字量对字符数组 a3 初始化
                             //因为文字量字符串有一个隐含的结束字符 '\0'
                             //因此 a3 实际是 4 个字符,即相当于 char a3[] = {'C','+',
                             //'+','\0'};
char a4[5] {"Hello"};        //错：空间不够,因为文字量字符串"Hello"实际有 6 个字符
char a5[6]{"Hello"};         //OK：空间正好
char a6[9]{"Hello"};         //OK：空间足够.问：a6 的第 8,9 个字符是什么呢
```

5.3.2 复杂的数组声明

因为数组本身是占据独立内存块的对象，所以可以定义指向它的指针或引用。

```
int ar[3];                   //3 个 int 类型元素的数组
int arr[10];                 //10 个 int 类型元素的数组
int *ptrs[10];               //10 个 int* 类型元素的数组
int (*parr)[10];             //parr 是一个指针,指向的是 int[10]的数组
                             //即指向的是 10 个 int 类型元素的数组
                             //或者说它存储的是 int[10]数组的地址
parr = &arr;                 //将 int[10]类型数组 arr 的地址赋值给 parr
parr = &ar;                  //错：类型不一致,ar 的类型是 int[3]而不是 int[10]
```

上述代码中 arr 的类型是 int[10]，arr 的地址（指针）类型是 int (*) [10]。理解上述变量的声明的方法是“**自内向外、自右向左**”。即首先从圆括号内看 parr 是一个指针（因为前面有一个星号 *），再从右边看，说明 parr 指向的是一个 10 个元素的数组，再向左看，说明数组中的元素类型是 int，即 parr 是一个指向 int[10]数组类型的指针。

正如定义指针变量 parr 指向一个数组 arr 一样，也可以定义一个引用变量引用一个数组。如：

```
int (&ref_arr)[10] = arr;    //ref_arr 是一个引用变量,引用的是 10 个 int 类型元素的数组
                             //而 arr 正好是 10 个 int 类型元素的数组
int (&ref_arr)[10] = ar;     //错：ref_arr 是一个引用变量
                             //引用的是 10 个 int 类型元素的数组
                             //而 ar 是 3 个 int 类型元素的数组。类型不一致
```

但不能定义一个“数据元素是引用的数组”，因为引用本身没有独立内存，怎么能定义这样的数组呢？

```
int &ref[10];                    //错：不能定义"数据元素是引用的数组"
```

上述代码的含义是：ref 是 10 个元素的数组，每个数组元素的类型是 int&，即 int 引用。不可能定义这样的数组，其中的每个元素没有独立存储空间。

5.3.3 C 风格字符串

带结束字符'\0'的字符数组是 C 语言的字符串，称为 **C 风格字符串**。但字符数组不一定是 C 风格字符串。

可以用<cstring>文件中的 C 字符串函数库处理 C 风格字符串，如 strlen(const char * s)可以求出一个 C 风格字符串中不包含结束字符的字符个数。

```
#include <iostream>
#include <cstring>
using namespace std;
int main(){
    char s[] = {'C','+','+'};              //字符数组，但不是C风格字符串
    char s2[] = {'C','+','+','\0'};        //带结束字符'\0'的字符数组是C风格字符串
    cout << strlen(s)<<'\t'<< strlen(s2)<< endl;
}
```

因为 s 不是 C 风格字符串，strlen(s)的结果是不确定的，如在作者计算机上运行程序后输出的结果是：

```
11 3
```

函数 strcmp(const char * s, const char * t)用来比较两个 C 风格字符串的大小，该函数返回的是一个 int 值：0 表示两个字符串完全一样；<0 表示第一个不匹配的字符 s 中的比 t 中的小；>0 表示第一个不匹配的字符 s 中的比 t 中的大。

例如：

```
#include <iostream>
#include <cstring>
using namespace std;
int main(){
    char s[] = "A string example";   //用字符串文字量初始化字符数组,结果包含了结束字符
    char s2[] = "A hello world";
    int ret = strcmp(s.s2);       //返回负数表示 s<s2,返回 0 表示 s==s2,返回正数,表示 s>s2
    if(ret < 0)
        cout <<"s < s2"<< endl;
    else if(ret == 0)
        cout <<"s == s2"<< endl;
    else
        cout <<"s > s2"<< endl;
}
```

表 5-1 列举了几个(不是全部)C 风格字符串处理函数。

表 5-1 几个 C 风格字符串处理函数

函 数 名	作 用
int strlen(const char * str)	求 str 的长度
char * strcpy (char * dst, const char * src);	将字符串 src 赋值给 dst,返回的是 dst
char * strcat (char * dst, cosnt char * src);	将字符串 src 拼接到 dst 的后面,返回的是拼接后的字符串即 dst
int strcmp (const char * str1, cosnt char * str2);	比较 str1 和 str2 的大小,返回值小于 0,表示 str1 < str2; 返回值等于 0,表示 str1 == str2; 返回值大于 0,表示 str1 > str2
const char * strchr (const char * str, int ch);	在字符串 str 中定位字符 ch 第一次出现的位置,返回这个字符的地址

函数形参的 const 修饰符的作用将在 5.5 节介绍。

例如,函数 strchr()用于查询一个字符串中是否出现某个字符并返回该字符的位置指针:

```
const char * strchr (const char * str, int character);
```

下面代码演示了 strchr()函数的用法:

```
/* strchr 例子 */
#include <iostream>
#include <cstring>
using namespace std;
int main (){
    char str[] = "This is a sample string";
    const char * pch;
    std::cout <<"字符 s 在字符串"<< str <<"出现的位置\n";
    pch = strchr(str,'s');
    while (pch!= 0) {
        std::cout <<"s 出现在"<< pch - str + 1 <<'\n';
        pch = strchr(pch + 1,'s');
    }
    return 0;
}
```

关于 C 风格字符串的更多函数和更详细信息,可以参考文档: http://www.cplusplus.com/reference/cstring/。

5.3.4 指针访问数组

数组名是指向数组第一个元素的指针(地址)。

```
int v[]{1,2,3,4};
int *p1{&(v[0])};          //p1 存储的是第一个元素 v[0]的地址
```

```
int *p2{v};              //数组名就是数组的第一个元素的地址,等价于 int * p2{&(v[0])};
int *p3{v + 4};          //v 是第一个元素的指针,向后偏移 4 个 int 类型元素空间
                         //因此 v + 4 指向的是最后一个元素 v[3]再偏移一个 int 类型元素空
                         //间的地址
std::cout << * (p1 + 2)<< endl;  //* (p1 + 2)等价于 * (v + 2),该语句输出第 3 个元素
```

如图 5-2 所示,对 int * 指针 v 加上一个整数 4,表示的是向后偏移 4 个 int 类型元素空间的地址。对指针也可以减去一个整数,表示向前偏移。实际上,用下标访问数组元素在编译过程中会转换成这种指针偏移。对整型变量 j,下列访问数组元素的式子都是等价的:

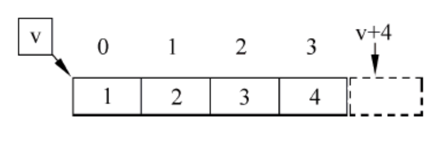


图 5-2 指针及其偏移

```
v[j] ==  * (&(v[0]) + j) ==  * (v + j)  ==  * (j + v)  == j[v]
```

例如:

```
3["hello"] == "hello"[3]
2[v]  =  v[2]
```

对于一个指针变量 p 和一个整数 n,除了可以用 p+n、p-n、p+=n、p-=n 等算术运算对指针进行偏移外,也可以用自增(p++或++p)、自减(p--或--p)运算进行偏移。例如:

```
int v[] = {1,2,3,4};
int * p = v;
p[2] = 20 + * (v + 3);
int b = * (p + 2),c = v[2],d = * (v + 2);
p++;                     //p 从 v 向后偏移一个 int 类型元素占据的空间(4 字节),即 p 指向 v[1]
p++;                     //p 指向 v[2]
std::cout << b <<'\t'<< c <<'\t'<< d <<'\t'<< * p <<'\n';
p -= 2;                  //p 向前偏移两个 int 类型元素占据的空间,即地址减去了 8 字节
                         //p 指向了第一个元素
std::cout << *p << std::endl;
p--;                     //语法不错,但其指向的地址已经不属于数组 v 了
std::cout << *p << std::endl;    //语法不错,但 *p 访问了一个不属于自己的内存块
                         //运行程序时会崩溃
int ival = 1024;
p = &ival;               //p 指向 ival
std::cout << *p << std::endl;    //*p 就是 ival
std::cout << * (p + 1)<< std::endl;    //语法没错,但 p + 1 指针指向的内存不属于程序
                         //* (p + 1)访问这个不属于自己的内存,运行时程序会崩溃
```

请体会下列程序用下标和指针访问数组元素的用法。

```
#include <iostream>
using namespace std;
int main(){
    int v[] = {1,2,3,4};
```

```
    for(int i = 0 ; i!= 4 ;i++)
        v[i]   = 2 * v[i] + 1;
    cout << endl;

    int *p = v;
    for(int i = 0;i!= 4; i++)
        cout << * (p + i)<<'\t';        //* (p + i)就是 p[i]
    cout << endl;

    for(int i = 0;i!= 4; i++)
        cout << p[i]<<'\t';             //p[i]就是 * (p + i)
    cout << endl;

    for(;p!= v + 4; p++)
        cout << *p <<'\t';
    cout << endl;

    p = v;
    int * q = v + 4;
    for(;p!= q; p++)
        cout << *p <<'\t';
    cout << endl;
}
```

两个指针不能相加，但指向同一个数组的指针可以相减，表示两者之间的元素个数。如：

```
int main(){
    int v1[10] ,v2[10];
    int a = &(v1[5]) - &(v1[3]);      //两个指针相隔两个整数,因此 a = 2
    int b = &(v1[5]) - &(v2[3]);      //不指向同一个数组的两个指针相减,
                                      //结果不可预知,因为这两个数组在内存的相对
                                      //位置不知道,可能相隔很远
    cout << a <<'\t'<< b <<'\t'<< endl;
    int *p = v1 + 11;
    a = p - &(v1[3]);                 //a 的值应该为 8
    std::cout << a << endl;
}
```

不指向同一个数组的同类型指针可以比较或相减，但没意义。再看一个例子。

```
int i = 0,sz = 42;
int *p = &i, *q = &sz;
int a = p - q;                        //不指向同一个数组的同类型指针相减,没有意义
if(p < q){                            //错: 这里比较没有意义
        //...
}
```

下列代码通过比较指向同一个数组元素的两个指针，控制循环过程：

```
int arr[10];
```

```
p = arr;  q = arr + 10;
a = q - p;                          //值为 10
a = p - q;                          //值为 - 10
while(p < q){
    std::cout << * p << endl;
    p++;                            //p 指向下一个 int 类型元素的位置
}
```

对于一个数组，可以用 C++标准库提供的 begin()和 end()函数得到这个数组的起始地址和结束地址（最后一个元素的后一个地址）。例如：

```
#include <iostream>
using namespace std;
int main() {
    int arr[]{ 1,2,3,4,5,6,7,8,9 };
    //begin(arr)返回 arr 的起始地址，相当于 arr 或 &(arr[0])
    //end(arr)返回 arr 的结束地址(即最后一个元素的后一个地址)，相当于 arr + 9 或 &(arr[9])
    auto b = begin(arr), e = end(arr);
    while (b != e) {
        cout << * b << '\t';          //通过解引用运算符 *，即 * b 得到 b 指向的 int 对象
        b++;
    }
    cout << endl;
}
```

其中 b、e 的类型都是 int * 。

5.3.5 range for

对数组这种多个同类型元素构成的序列对象，可以用 **range for** 遍历其中的元素。有下面两种形式（假如 arr 是 T 类型的数组）：

```
//假如 arr 是 T 类型的数组
for(T & 变量名: arr)                //变量名表示是 arr 每个元素的引用
```

或

```
for(T 变量名: arr)                  //变量名表示是 arr 每个元素的复制
```

例如：

```
#include <iostream>
int main(){
    int arr[] = {1,2,3,4,5,6,7,8,9};
    //访问 arr 的每个元素，变量 e 引用该元素
    for(int &e: arr)        //对 arr 的每个元素 e，可以通过引用变量 e 直接修改数组的元素
        e = 2 * e + 1;
```

```
    //访问 arr 的每个元素,用该元素初始化变量 e
    for(int e: arr)         //e 是 arr 元素的赋值(复制),无法通过 e 修改 arr 的元素
        std::cout << e <<'\t';
    std::cout << std::endl;
}
```

当执行这种 for 语句时，将依次迭代(循环)访问数组 arr 的每个元素，可以用一个引用变量去引用数组元素，也可以用值变量得到数组元素的复制。然后进行相应的处理。

当然，可以将 range for 的变量类型用 auto 来自动推断，即：

```
#include <iostream>
int main(){
    int arr[] = {1,2,3,4,5,6,7,8,9};
    //访问 arr 的每个元素,变量 e 引用该元素
    for(auto &e: arr)       //对 arr 的每个元素 e,因为要修改 arr 的每个元素
                            //所以 e 必须定义为是引用变量
        e = 2 * e + 1;
    //访问 arr 的每个元素,用该元素初始化变量 e
    for(auto e: arr)        //对 arr 的每个元素 e,不修改 arr 的元素
                            //所以 e 可以定义成非引用变量
        std::cout << e <<'\t';
    std::cout << std::endl;
}
```

注意：range for 不能用于指针。例如：

```
int is[] = {1,2,3,4};
for (auto e: is)
    std::cout << e <<'\t';
std::cout << std::endl;

int *p = is;                //p 是指针数组 is 的指针变量
for (auto e: p)             //错: 不能将 range for 用于指针 p
    std::cout << e <<'\t';
std::cout << std::endl;
```

5.3.6　多维数组

严格地说，C++没有提供多维数组，只有一维数组。所谓的多维数组是通过一维数组来表示的。也就是说多维数组实质上就是一维数组，只不过这个一维数组的元素仍然是一个数组，并且可以一直这样表示下去。例如：

```
int ia[3][4];               //ia 是 3 个元素的一维数组,其中每个元素(ia[0],ia[1],ia[2])
                            //是一个 int[4]的一维数组,即 4 个 int 类型元素的数组
```

C++的**多维数组本质上是一维数组**，这对于理解 C++的多维数组非常重要。

按照“**由内向外、自右向左**”的阅读方法，ia 是一个 3 个元素的数组，而每个元素又是一

个 int[4]的数组,即每个元素是一个包含 4 个 int 类型元素的数组。再如:

```
int arr[10][20][30] = {1};
```

可以按如下来理解变量 arr。

- (从内到外)arr 是 10 个元素的数组,其每个元素的类型是 int[20][30]。
- (从内到外)int[20][30]又是 20 个元素的数组,其每个元素的类型是 int[30]。
- (自右往左) int[30]又是 30 个元素的数组,其每个元素是一个 int 类型对象。

列表初始化{1}中只有一个元素,是对 arr 的第一个元素(第一行)即 arr[0]初始化,进而对 arr[0]的第一个元素 arr[0][0]初始化,进而对 arr[0][0]的第一个元素 arr[0][0][0]初始化。其他的元素就取默认值 0。

再看一个例子:

```
int ia[3][4] = {          //3 个元素的数组,每个元素又是一个 int[4]的数组
    {0,1,2,3},            //对 ia 的第 1 个元素 ia[0]初始化
    {4,5,6,7},            //对 ia 的第 2 个元素 ia[1]初始化
    {8,9,10,11}           //对 ia 的第 3 个元素 ia[2]初始化
};
//也可以等价地写成
int ib[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
//列表初始化,如果提供的初始化值个数不足,就取默认值
int ic[3][4] = {{0},{4},{8}}; //每一行的第 1 个元素的初始值
int ic[3][4] = {0,4,8};       //第 1 行的前 3 个元素的初始值
```

同样,可以用下标运算符[]访问多维数组的元素,例如:

```
ia[2][3] = arr[0][0][0];
int (&row)[4] = ia[2];    //int[4]类型的引用变量 row
                          //引用 ia 的第 3 个元素,即第 3 行
```

再看下面完整的例子。

```
#include <iostream>
using namespace std;
int main(){
    int ia[3][4];         //= {0,1,2,3,4,5,6,7,8,9,10,11};
    for(int i = 0; i!= 3; i++)
        for(int j = 0; j!= 3; j++)
            a[i][j] = i * 4 + j;

    //演示如何用引用访问数组
    int (&row)[4] = ia[0];//row 引用 ia 的第 1 行
    for(int j = 0; != 4; j++)
        cout << row[j]<<'\t';

    //演示如何用指针遍历数组
    int ( *p)[4];         //指针变量 p 指向的是一个 int[4]数组
```

```
    p = ia;              //ia 即数组 ia 的第 1 个元素的地址,即 &(ia[0]),因此,p 存储的是
                         //ia[0]的地址
    for(; p!= ia + 3 ;p++){     //p 指向的是 int[4]数组, *p 就是它指向的 int[4]数组
        int *q = *p;     //q 指向 *p 这个数组的第 1 个元素
        for(; q!= (*p) + 4; q++)
            cout << *q <<'\t';
        cout <<'\n';     //输出换行
    }
}
```

请仔细体会代码中的引用和指针的用法。

这种引用和指针的用法对初学者来说理解比较困难,为什么不用 C++11 以后提供的更加简单的 auto 和 rang for 呢?

先将上述代码用 auto 来改写一下:

```
//ia 数组名就是数组的第 1 个元素的地址,因此 p 是指向 ia[0]的指针
for (auto p = ia; p != ia + 3; p++) {
    //*p 是一个 int[4]数组,当然也就是这个数组(第 1 个元素)的地址
    //q 的类型自动推断为 int *
    for (auto q = *p; q != *p + 4; q++)
        cout << *q << '\t';
    cout << '\n';          //输出换行
}
```

还可以使用 begin()和 end()函数得到数组的起始和结束位置。

```
for (auto p = begin(ia); p != end(ia); p++) {
    for (auto q = begin( *p); q != end( *p); q++)
        cout << *q << '\t';
    cout << '\n';          //输出换行
}
```

这样便不需要在代码里以硬编码的方式给出结束位置,不但更具有通用性,也不容易出错。

还可以用 range for 写出更简单的代码,如下:

```
using namespace std;
int main() {
    int ia[3][4];
    auto cnt{ 0 };
    for (auto &row : ia)              //对 ia 的每个元素(每行)的引用 row
        for (auto &col : row) {       //对 row 的每个元素(每列)的引用 col
            col = cnt;    cnt++;      //可以合并为一句: col = cnt++;
        }
    //输出
    for (auto &row : ia) {
        for (auto &col : row)
            cout << col << '\t';
        cout << '\n';
```

```
    }
}
```

执行程序,输出结果:

```
0        1        2        3
4        5        6        7
8        9        10       11
```

注意:将 row 和 col 定义成引用变量,即直接引用原来数组元素而不是复制它们的值。假如编写下面的代码:

```
//row 用 ia 的每个元素初始化,即每个元素的值复制给 row
//元素 ia[i]是一个数组,会自动转换为指针,即 row 实际是一个指针类型 int ( * )[4];
//也就是说 row 是一个指向 int[4]的指针
for (auto row : ia)
    for (auto col : row)              //错: row 是一个指针
                                      //而对于指针是无法使用 range for 的
        cout << col << '\t';
```

因为对于指针,是无法使用 range for 的,上述代码将产生编译错误。

因此,除了最内层外,其他层的 rang for 元素的变量必须声明为引用类型,如下所示:

```
for (auto& row : ia) {          //row 是引用 ia 的每个元素,因此 row 是一个 int[4]的数组
    for (auto col : row)
        cout << col << '\t';
```

5.4 动态内存

前面的所有变量占用的内存都是静态分配的,编译器在编译时就为每个变量分配了固定大小的内存。例如,C++的内在数组定义时就必须说明数组的大小,即数组元素的个数,编译器才能为这个数组的所有数据元素分配一块固定大小、连续的内存块。程序运行期间,数组的大小不能改变,这带来两个问题:程序在某些情况下可能会出现数组空间不足的问题,如一个只能存放 100 个学生的程序无法用于超过 100 个学生的情形;如果数组很大,又会造成空间浪费的问题,如分配 5000 个学生的数组,而实际情况下,学生人数通常不超过 100。为了解决这种空间不足或浪费的问题,可以用 C++提供的动态内存分配功能,即在程序运行过程中,根据实际需要分配相应大小的内存。

5.4.1 程序堆栈区

每个程序在计算机中都占用一块内存,这块内存用于存放程序的代码和数据,每个程序除了代码占据的内存外,都有一个称为**堆栈(stack)**的内存块,用于存储程序块的非静态局部变量。当进入一个程序块时,这个程序块中(非静态)局部变量就在堆栈的顶部分配一块内存,称为**变量入栈**;当退出这个程序块时,这个程序的(非静态)局部变量在栈顶的内存就

被释放，称为**变量出栈**。

对于如下代码：

```
int main() {
    int a{ 3 }, b{ 4 };
    {
        int c{ 5 };
    }
    a = 2;
}
```

如图 5-3 所示，在程序进入 main()主函数时，即 main()主函数的{}定义的程序块时，main 程序块的局部变量 a、b 入栈，当进入内部{}定义的程序块时，这个新的程序块的局部变量 c 入栈，即创建了 c(给 c 在栈顶分配了内存)，当退出这个程序块时，c 出栈，即 c 被销毁。当执行完 main()的最后一条语句“a=2”退出 main()函数时，栈顶的 a、b 也出栈(即 a、b 被销毁)。

```
int main() {
    int a{ 3 }, b{ 4 };   ←
    {
        int c{ 5 };
    }
    a = 2;
}
```



```
int main() {
    int a{ 3 }, b{ 4 };
    {
        int c{ 5 };   ←
    }
    a = 2;
}
```



```
int main() {
    int a{ 3 }, b{ 4 };
    {
        int c{ 5 };
    }
    a = 2;   ←
}
```


图 5-3　程序堆栈存储程序块的局部变量

5.4.2　new 和 delete 运算符

计算机的内存除了被分配给正在运行的多个程序(包括操作系统程序)的内存外，剩余的空闲内存称为**自由内存**或**堆存储区**(简称**堆区**)，**堆区**是所有程序共享的自由存储区。任何程序都可以向操作系统申请这块堆区的一块内存。C++中可通过 **new** 运算符向操作系统申请堆区的一块内存，并通过 **delete** 运算符释放这块内存(即将内存还给操作系统)。这种通过 **new** 和 **delete** 申请和释放堆区的内存的过程称为**动态内存分配和释放**。

对于一个数据类型 T，**new** T 用于申请一个 T 类型大小元素的内存，而 **new** T[size]用于申请可存储 size 个 T 类型元素的一块内存。new T 和 new T[size]都返回分配内存块的

起始地址，返回值的类型是 T *，即指向 T 指针类型。如果 new 申请内存失败，返回的值是 0。例如：

```
int main() {
    double *p{nullptr };                //初始化指针变量 p 为空指针
    p = new double;                     //p 存储 new 分配内存的地址
    if(!p) return 1;                    //申请内存失败
    double *q{nullptr };
    q = new double[3];                  //q 存储 new 分配内存的地址
    if(!q) return 1;                    //申请内存失败

    //可以用解引用运算符 * 访问 p 或 q 指向的内存
    *p = 3.14;                          //*p 就是 p 指向的那个 double 变量
    *q = 3.15;                          //*q 就是 q 指向的那个 double 变量
    *(q + 1) = 3.16;                    //*(q+1)就是 q+1 指向的那个 double 变量
    *(q + 2) = 3.17;                    //*(q+2)就是 q+2 指向的那个 double 变量
    *(q + 3) = 3.17;                    //*(q+3)就是 q+3 指向的那个 double 变量
                                        //但 q+3 指向的内存不属于分配的内存块,无法访问,
                                        //因此出错

    *(p + 1) = 2.5;                     //错: p+1 的内存不属于程序
    *(q - 1) = 2.5;                     //错: q-1 的内存不属于程序
}
```

上述程序分配了 2 块内存，然后可以通过**解引用运算符** * 访问指针指向的内存。对指针可以加减整数进行偏移，但如果超出了分配内存块的范围，则访问非法。

对一个指针 p，因为 p[i]就是 *(p+i)，因此当然可以通过下标访问指针指向的动态内存：

```
int main() {
    …
    for (auto i = 0; i < 3; i++)
        q[i] += 2;
    for (auto i = 0; i < 3; i++)
        std::cout << q[i] << '\t';
}
```

执行程序，输出结果：

```
5.15    5.16    5.17
```

对于动态分配的内存，当不再使用时，应该及时释放，以便程序的其他部分或其他程序能使用这块内存。一个或多个程序如果不断申请动态内存而没有及时释放，就会使自由内存越来越少，最后会导致内存耗尽而使程序无法运行。假设指针变量 p 指向动态内存的起始地址，对于 new T 分配的一个 T 元素的内存，用 delete p 释放 p 指向的这块 T 元素占据的内存。对于 new T[size]分配的多个 T 元素空间的内存，用 delete[] p 释放 p 指向的多个 T 元素占用的内存，如果写成了 delete p，释放的将是第一个 T 元素占用的内存，其他元素

的内存并没有得到释放，这会造成**内存泄漏**。

```
delete p;
delete[] q;  //如果写成 delete q;会造成内存泄漏
```

释放 p 指向的内存块后，应该将 p 设置为空指针“p = nullptr;”。这是一个良好的编程习惯，可以避免多次释放同一块内存或访问一个已经释放的内存而导致程序崩溃。

5.4.3　动态内存表示多维数组

假如学生成绩分析程序中每个学生的成绩不是一个值而是多个值，包含平时成绩、实验成绩、期末成绩、总评成绩，一个班级所有学生的成绩可以用一个二维数组表示：

```
double scores[100][4];           //最多可以存储 100 个学生的成绩
int n = 0;                       //学生人数
```

那么是否可以直接用动态内存来表示这种二维数组呢？如写成下面的代码：

```
int n = 0;                       //学生人数
int cols;                        //每个学生的成绩个数
std::cin >> n >> cols;
double * scores = new double[n][cols];
```

答案是否定的。new 只能分配一维的一组数据元素而不能直接分配二维数组这种动态空间，这是因为内存本身就是一维的。

前面说过，C++内存的多维数组实质也是一维数组，即 double scores[100][4]其实是一个大小为 100 的一维数组，每个元素的类型是 double [4]，即每个元素也是一个大小为 4 的一维数组。因此，可以用 new 分配类型是 double [4]的一组数据元素空间，即：

```
double ( * scores)[4] = new double[n][4];
```

当然，可以用 auto 简化指针类型的声明：

```
auto scores{ new double[n][4]};
```

注意：因为 4 是文字常量，所以 double [4]就是一个编译时大小确定的数组类型，当然可以用 new 申请这种类型的一维数组 new double[n][4]。而对于变量 cols，double [cols]并不是一个大小确定的数据类型，所以不能用 new 申请这种类型的一维数组空间，即 new double[n][cols]是非法的。

可以编写如下基于动态内存分配的学生成绩分析程序框架：

```
#include <iostream>
int main() {
    int n = 0;                   //学生人数
    int cols;                    //每个学生的成绩个数
```

```
    std::cout << "输入学生人数\n";
    std::cin >> n ;
    auto scores{ new double[n][4]};
    std::cout << "输入学生的平时、实验、期末、总评成绩\n";
    for (auto i = 0; i != n; i++) {
        std::cin >> scores[i][0] >> scores[i][1]
            >> scores[i][2] >> scores[i][3];
    }
    for (auto i = 0; i != n; i++) {
        std::cout << scores[i][0] <<'\t'<< scores[i][1]  << '\t'
            << scores[i][2] << '\t'<< scores[i][3] << '\n';
    }
}
```

执行程序,输出结果:

```
输入学生人数
3
输入学生的平时、实验、期末、总评成绩
80 60 50 0
78 80 90 0
85 60 70 0
80      60      50      0
78      80      90      0
85      60      70      0
```

5.5 const 修饰符

第 2 章说过,const 修饰基本类型的变量时,表示这个变量是不可修改的。const 和复合类型结合,其含义就不那么简单了,需要根据 const 在变量声明中的位置来理解其含义。

5.5.1 const 和指针

下列代码定义了 3 个指针变量 p、q、s:

```
int i{0};
int * const p = &i;
const int *q = &i;
int const *s = &i;
std::cout << *p << '\t'<< *q << '\t'<< *s << '\t';
```

根据理解变量的从右向左规则,紧挨着 p 的是一个 const,说明 p 首先是一个 const 变量(对象),即 p 的值是不可以修改的,再往左看,是 int * 说明 p 变量存储的是 int * 的值,即 int 变量的地址,即 p 是不可被修改的 int * 指针变量,或者说 p 是一个 int * 的 const 对象。

q 和 s 紧挨着的是 *,说明它们首先是一个指针变量,再往左看是 const int 和 int

const，说明 q 和 s 指向的变量是 const int 和 int const。const int 和 int const 的类型是一样的，因此，q 和 s 的类型是一样的，是指向 const int 变量的指针变量，即它们指向的变量是不能被修改的(const int)，但它们本身是可以被修改的。而前面的 p 是不可修改的，但 p 指向的 int 变量是可以修改的。

p 称为 **const 指针**，即不能被修改的指针，而 q 和 s 称为 **const 对象的指针**，即其指向的 const 对象不能被修改。

因此，下列代码是没有任何错误的：

```
*p = 2;             //p不能被修改,但它指向的 int 类型变量是可以被修改的
std::cout << *p << '\t'<< *q << '\t'<< *s << '\n';
int j{ 3 };
q = &j;             //修改 q 指向另一个 int 变量 j
s = &j;             //修改 s 指向另一个 int 变量 j
std::cout << *p << '\t'<< *q << '\t'<< *s << '\n';
```

执行后的结果：

```
2       2       2
2       3       3
```

修改 p、修改 q 或 s 指向的变量都是非法的：

```
p = &j;             //错: p 是不能被修改的
*q = 4;             //错: q 指向的是 const int 对象,const 对象是不能被修改的
*s = 4;             //错: s 指向的是 const int 对象,const 对象是不能被修改的
```

VS2017 显示的语法错误：

```
…: error C3892: "p": 不能给常量赋值
…: error C3892: "q": 不能给常量赋值
…: error C3892: "s": 不能给常量赋值
```

当然，还可以定义指针变量如下：

```
const int * const ptr = &i;
```

ptr 是 const 指针，且它指向的也是一个 const 对象。因此，不但 ptr 不能被修改(必须始终指向 i)，而且其指向的 const int 变量也不能被修改。

指针和 const 可产生如下组合：

```
char * const cp;            //指向 char 的 const 指针
char const * pc;            //指向 const char 的指针
const char * pc2;           //指向 const char 的指针
const char * const pc3;     //指向 const char 的 const 指针
```

即 cp 和 pc3 指针变量的值都不能被修改(即都是 **const 指针**)，但 cp 指向的是 char 对

象的指针，即 cp 指向的 char 对象是可以被修改的，而 pc3 指向的是 const char 对象，那么不能修改 pc3 指向的变量，即 pc3 也是 const 对象的指针。而 pc 和 pc2 都是可以修改的指针，但它们指向的是 const char，因此，它们是指向 **const 对象的指针**。

有的书上将 const 指针称为常指针(如 cp、pc3)，而一个指针指向的如果是 const 对象，则称为常量的指针(如 pc、pc2)。当然 pc3 既是常指针也是常量的指针，可称为常量的常指针。

可通过下面的代码进一步加深理解：

```
int a = 0;
const int b = a;
const int * pa = &a;
 * pa = 4;                      //错: 指向 const 对象
pa = &b;                        //指针可以修改
int * const pa2 = &a;
pa2 = &b;                       //错: const 指针不能被修改
int * pb = &b;                  //错: 普通指针不能指向 const 对象
                                //否则,不就可通过 *pb 修改 const 对象了
const int * pb2 = &b;
pb2 = &a;
const int * const pb3 = &b;
const int * const pb4 = &a;
pb4 = pb3;                      //错: pb4 不能被修改
 * pb4 = 9;                     //错: pb4 指向的 const 对象,不能被修改
```

再如：

```
char s[] = "Good";
const char * pc = s;            //指向 const char 的指针
pc[3] = 'g';                    //错:不能通过 pc 修改它指向的 const char
pc = p;                         //pc 本身是可以被修改的
char * const cp = s;            //指向 char 的 const 指针
cp[3] = 'a';                    //OK,cp 指向的 char 变量可以被修改
cp = p;                         //错: cp 是 const 指针,不能被修改
const char * const cpc = s;     //指向 const char 的 const 指针
cpc[3] = 'a';                   //错: cpc 指向 const char
cpc = p;                        //错: cpc 是 const 指针
```

希望读者能好好体会常量的指针(pointer to constant)和常指针(constant pointer)的区别。

5.5.2 const 对象的引用

类似于 const 对象的指针，可以定义 const 对象的引用。即引用变量绑定的是一个 const 对象。既然是一个 const 对象的引用，就不能通过该引用变量去修改它引用的对象。

const 对象的引用可以用非 const 对象、文字量和一般表达式初始化。例如：

```
int i = 42;
const int ci = 1024;            //ci 是 const int 对象,即不能修改,也称为常量
```

```
const int &r1 = ci;          //用 const int 初始化 const int 的引用变量 r1
const int &r2 = i;           //用 non-const 的 int 变量 i 初始化 const int 的引用变量 r2
const int &r3 = 42;          //用文字量 42 初始化 const int 的引用变量 r3
const int &r4 = r1 * 2;      //用表达式 r1 * 2 初始化 const int 的引用变量 r4
```

可以用“非 const 对象”或“表达式”初始化一个 const 对象的引用，只要这个表达式类型能转换成引用的类型。

const 对象的引用往往绑定的是一个临时变量。如：

```
double dval = 3.14;
const int &r8 = dval;        //dval 是 double 类型,而 r8 是 const int 类型的引用
```

r8 实际上是绑定到一个临时变量而不是 dval。即编译器实际上创建了一个临时变量，即将“const int &r8 = dval;”替换为如下形式：

```
const int temp = dval;
const int &r8 = temp;
```

反过来，不能用 **const 对象**、**文字量**、**表达式**初始化一个 non-const(非 const 对象)的引用。例如，下面的代码是错误的：

```
int &r5 = ci;                //错: 普通变量的引用不能引用 const 对象
int &r6 = i * 2;             //错: 普通变量的引用不能引用表达式
int &r7 = 6;                 //不能引用文字量
```

试图修改 const 对象的引用是非法的：

```
r1 = 42;                     //错: const 对象的引用不能用于修改
r4 = 42;
```

因此，不能通过 const 对象的引用去修改其绑定的对象，即使原来的对象实际是可修改的。

```
int j = 4;
int &r9 = j;
const int &r10 = j;          //r10 是 const int 的引用,实际上绑定的是一个临时变量,而不是 j
r9 = 0;                      //也就是 j = 0
r10 = 0;                     //不可以,因为 r10 是 const int 的引用
```

5.6 实战：查找、排序、最短路径

5.6.1 二分查找

1. 顺序查找

要查找一个序列中是否存在某个元素，可以采用和序列的每个元素依次比较的方法，如

依次和序列的第 1 个、第 2 个、第 3 个、……、最后一个元素比较。这种查找方法称为顺序查找。

```
#include <iostream>
using std::cout;
int main() {
    int a[]{ 12,46,25,43,7,92,5,29,80,105 };
    auto x{ 0 };
    bool found{ false };
    std::cin >> x;
    for (auto e : a)
        if (e == x) {
            found = true;
            break;
        }
    if (found)
        std::cout << "在数组 a 中找到了: " << x <<'\n';
    else
        std::cout << "在数组 a 中未找到: " << x <<'\n';
}
```

该程序用一个 range for 循环依次遍历序列 a 的每个元素，和要查找的元素 x 比较。如果有一个元素满足 e==x，则设置标志 found 为 true，并跳出循环。

执行程序，如果输入一个值 25，则程序运行情况为：

```
25
在数组 a 中找到了: 25
```

执行程序，如果输入一个值 13，则程序运行情况为：

```
13
在数组 a 中未找到: 13
```

分析这个程序的时间效率：如果要查找的元素是第 1 个，则只要 1 次比较，如果是第 2 个元素，则要 2 次比较，……。假设一个序列中有 n 个数据元素，每个元素的查找概率是均等的 $1/n$，那么查找成功情况下的平均比较次数是：$1*(1/n)+2*(1/n)+\cdots+n*(1/n)=(n+1)/2$，即查找成功情况下，平均需要比较的次数几乎是表长的一半。当 n 趋向于无穷大时，$(n+1)/2$ 和 n 是同一个数量级，所以称**其时间复杂度**是 $O(n)$，即查找时间和 n 是同介无穷大量。

假如 $n=1024$，成功查找其中的一个元素平均需要比较 512 次。但如果这个序列是有序的，那么可以使用一种“**二分查找**”的算法，平均只需要 $\log_2(1024)$次，即 10 次就可以找到这个元素。

2. 二分查找

二分查找的思想很简单，对一个有序序列 a，要查找某个元素 x，则可以让 x 先和 a 的中间元素比较：

- 如果相等，则成功。
- 如果小于中间元素，则在 a 的左半区间查找。
- 如果大于或等于中间元素，则在 a 的右半区间查找。

图 5-4 是在一个有序序列中查找 25 的过程，其中 L、H、Middle 分别表示当前查找区间的左、右和中间位置。

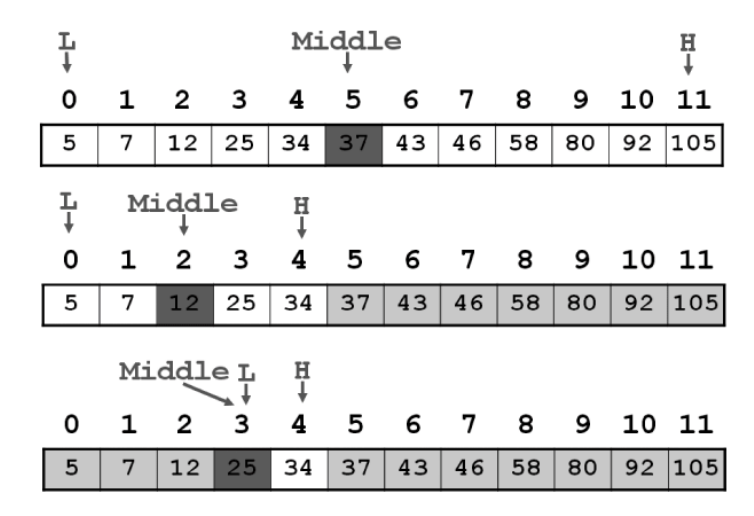


图 5-4 二分查找 25 的过程

可用一个循环迭代过程，不断更新区间的左、右位置（L、H）和中间位置（Middle）三个指示器来查找。

```
#include <iostream>
using std::cout;
int main() {
    int a[]{ 5,7,12,25,34,37,43,46,58,80,82,105 };
    auto x{ 0 };
    std::cin >> x;

    auto L{ 0 }, H{ 9 };
    while (L <= H) {                        //区间[L,H]存在
        auto Middle{ (L + H) / 2 };         //Middle 指向区间的中点
        if (a[Middle] == x)                 //等于 Middle 指向的元素，找到了
            break;
        else if (x < a[Middle])
            H = Middle - 1;                 //在左区间查找
        else
            L = Middle + 1;                 //在右区间查找
    }
    if(L <= H)
        std::cout << "在数组 a 中找到了：" << x << '\n';
    else
        std::cout << "在数组 a 中未找到：" << x << '\n';
}
```

执行程序，如果输入的值为 34，输出结果：

```
34
在数组 a 中找到了: 34
```

5.6.2 排序：冒泡、选择

所谓排序就是将一个序列按照数据元素(或其关键字)的大小重新排序，使得所有元素按照“从小到大”或“从大到小”的次序排列在这个序列中。

如一个数值的序列 (7,2,9,4,6)，按照从小到大的顺序排序可得到一个新的序列：(2,4,6,7,9)。

排序的应用很广，如对一组学生，可以按照身高、成绩、姓名等排序。搜索引擎中对搜索得到的信息可以按照其价值、时间等排序。各种排行榜也按照某种标准排序，如每年不同机构或组织的编程语言排行榜、高校排行榜等。

下面介绍几个简单的排序算法。

1. 冒泡排序

冒泡排序的思想是：对相邻元素比较大小，如果逆序就交换它们。如图 5-5 所示，对一个序列，通过这种两两相邻元素的比较与交换，可以将最大值放在最后一个位置，这一过程称为**“一趟冒泡”**。

j=0，交换	49	38	27	97	76	13	27	49
j=1，交换	38	49	27	97	76	13	27	49
j=2，不交换	38	27	49	97	76	13	27	49
j=3，交换	38	27	49	97	76	13	27	49
j=4，交换	38	27	49	76	97	13	27	49
j=5，交换	38	27	49	76	13	97	27	49
j=6，交换	38	27	49	76	13	27	97	49
j=7，结束	38	27	49	76	13	27	49	97

图 5-5 “一趟冒泡”选择一个最大值放在目标位置

“一趟冒泡”只是在一个序列中选出了一个最大值并放在了其最终位置，对于剩余元素构成的序列，再进行“一趟冒泡”，又可以在剩余元素序列中选出一个最(大)值并放在其最终位置(如倒数第二个位置)。重复这个过程，直到剩余一个元素。对于 n 个元素的序列，需要进行 $n-1$ 趟冒泡。

```
#include <iostream>
int main() {
    int a[]{ 49, 38, 27, 97, 76, 13, 27, 49 };
    for (auto i = 7; i > 0; i-- ) {       //i 从 7 到 1,共 7 趟冒泡
        //对每个 i,对[0,i]的序列进行"一趟冒泡"
        for (auto j = 0; j < i; j++)      //下标 j 遍历序列[0,i-1]
```

```
                if (a[j] > a[j + 1]) {        //如果是逆序,就交换它们
                    auto t = a[j]; a[j] = a[j + 1]; a[j + 1] = t;
                }
            //输出序列
            for (auto e : a)
                std::cout << e << '\t';
            std::cout << '\n';
        }
    }
```

执行程序，输出结果：

```
38      27      49      76      13      27      49      97
27      38      49      13      27      49      76      97
27      38      13      27      49      49      76      97
27      13      27      38      49      49      76      97
13      27      27      38      49      49      76      97
13      27      27      38      49      49      76      97
13      27      27      38      49      49      76      97
```

上述程序也可以用指针代替下标访问数组的元素，代码如下：

```
#include <iostream>
int main() {
    int a[]{ 49, 38, 27, 97, 76, 13, 27, 49 };
    for (auto r = a+7; r>a; r-- ) {
        //对[0,i]的序列进行"一趟冒泡"
        for (auto p = a; p<r; p++)
            if ( *p> *(p + 1)) {        //交换它们
                auto t = *p; *p = *(p+1); *(p+1) = t;
            }
        //输出序列
        for (auto e : a)
            std::cout << e << '\t';
        std::cout << '\n';
    }
}
```

2. 简单选择排序

简单选择排序的思想是：在整个序列中选出一个最值(如最小值)放在序列的开头(或结束)位置。这个过程称为“**一趟选择**”。对于剩余元素构成的序列重复这个过程，又选出一个最值放在剩余元素序列的开头(或结束)位置，即“第 2 趟选择”。这个过程一直进行下去，直到剩余序列只有一个元素。请读者根据这个思想写出简单选择排序的程序。

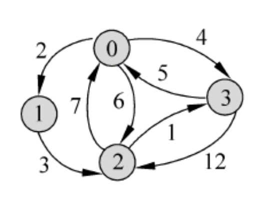
5.6.3　Floyd 最短路径算法

1. 最短(最佳)路径问题

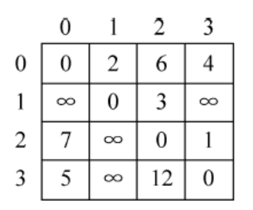
最短路径问题是日常生活和科学研究中广泛应用的问题，例如，一个人在一个陌生城

市，要从某一个地点到另外一个地点，会打开手机地图导航软件，软件会按不同代价(时间、路程、花费)给出不同的最佳(最短)路径。再如计算机网络通信，需要在不同计算机之间发送数据包，网络路由算法会采用最短路径算法计算出最佳的数据包发送路径。

假如图 5-6(a)表示的是一个城市公路图，其中顶点表示城市，而边表示城市之间的公路距离，现在要求出任何两个城市之间的最短路径。最短路径属于图论的一个基本问题，针对这个问题，有很多最短路径算法，其中的 Floyd 算法可以求出任意两个顶点之间的最短路径。



(a) 某城市公路图



	0	1	2	3
0	0	2	6	4
1	∞	0	3	∞
2	7	∞	0	1
3	5	∞	12	0

(b) 邻接矩阵

图 5-6 某城市公路图及其邻接矩阵

2. Floyd 算法

Floyd 算法的基本思想是：用一个二维矩阵表示任意 2 个顶点之间的距离，如图 5-6(b)所示。初始时，这个矩阵的数据元素的值表示的是 2 个城市的直达距离。这个初始的距离矩阵也称为**邻接矩阵**。

直达距离不一定是最短距离(如 0 到 2 的直达距离是 6，但 6 不是 0 到 2 的最短距离)。为求任意两个顶点之间的最短距离，Floyd 算法每次考虑绕道一个顶点，看看是否有两个顶点的距离会因为绕道这个顶点变得更短。

假如当前的距离矩阵为 D(初始时 D 就是邻接矩阵)，现在绕道顶点 w，看看顶点 u 到 v 之间的距离 D[u][v]会不会因为绕道这个顶点 w 变得更短，即 D[u][w]+ D[w][v] < D[u][v]是否成立。如果更短，则更新 D[u][v]为 D[u][w]+ D[w][v]。

```
if(D[u][w]+ D[w][v] < D[u][v])
      D[u][v] = D[u][w]+ D[w][v] ;
```

每绕道一个顶点，就可能会更新这个距离矩阵的某两个顶点的最短距离。对每个顶点都重复这个绕道过程，直到所有顶点都绕道完为止，最终得到的矩阵就记录了任意两个顶点的最短距离。

为了记录路径，还需要一个和 D 一样大小的二维矩阵 P，用来记录任意两个顶点之间的当前距离对应的路径上的倒数第二个顶点(即路径上终点之前的那个顶点)。当绕道顶点 w，使得 u 到 v 的距离更短时，不但更新距离矩阵，还更新这个路径矩阵。

```
if(D[u][w]+ D[w][v] < D[u][v]){
      D[u][v] = D[u][w]+ D[w][v];
      P[u][v] = P[w][v];       //P[u][v] = P[u][w]+ P[w][v]
}
```

“P[u][v] =P[w][v];”表示 u 到 v 的最短路径上的终点(终点当然就是 v)的前一个顶点就是绕道 w 到达 v 的路径上的那个“终点的前一个顶点”,即 P[w][v]。

下面是一个完整的最短路径程序代码：

```
#include <iostream>
using namespace std;
int main() {
    auto n{ 4 };
    auto INFINITY =  std::numeric_limits<double>::max();
    //auto INFINITY = 1e12;        //INFINITY 假设表示一个无穷大的数值
    //初始化距离矩阵 D
    double D[ ][4]{ {0, 2, 6, 4},
                {INFINITY, 0, 3, INFINITY},
                {7, INFINITY, 0, 1 },
                {5, INFINITY, 12, 0} };
    //初始化路径矩阵 P
    unsigned int P[ ][4]{ {0, 0, 0, 0},{0, 0, 0, 0},{0, 0, 0, 0},{0, 0, 0, 0} };
    for (auto u = 0; u < n; u++)
        for (auto v = 0; v < n; v++)
            P[u][v] = u;          //直达路径 uv 的终点的前一个顶点是 u,终点是 v
    //打印 D 和 P 矩阵
    cout <<"初始的距离和路径矩阵: \n";
    for (auto& row : D) {         //row 是引用 D 的每个元素,因此 row 是一个 int[4]的数组
        for (auto col : row)
            cout << col << '\t';
        cout << '\n';
    }
    cout << '\n';
    for (auto& row : P) {         //row 是引用 p 的每个元素,因此 row 是一个 int[4]的数组
        for (auto col : row)
            cout << col << '\t';
        cout << '\n';
    }
    cout << std::endl;

    //Floyd 算法
    for (auto w = 0; w < n; w++)  //对每个顶点 w,都绕道一次,更新 D 和 P
        for (auto u = 0; u < n; u++)
            for (auto v = 0; v < n; v++)
                //绕道 w 使 D[u][v]变得更短吗
                if (w != u and w != v and D[u][w] + D[w][v] < D[u][v]) {
                    D[u][v] = D[u][w] + D[w][v];
                    P[u][v] = P[w][v];
                }
    //打印 D 和 P 矩阵
    cout <<"最终的距离和路径矩阵: \n";
    for (auto& row : D) {
        for (auto col : row)
```

```
            cout << col << '\t';
        cout << '\n';
    }
    cout << '\n';
    for (auto& row : P) {
        for (auto col : row)
            cout << col << '\t';
        cout << '\n';
    }
    cout << std::endl;
}
```

执行程序,输出结果:

```
初始的距离和路径矩阵:
0       2       6       4
inf     0       3       inf
7       inf     0       1
5       inf     12      0

0       0       0       0
1       1       1       1
2       2       2       2
3       3       3       3
最终的距离和路径矩阵:
0       2       5       4
9       0       3       4
6       8       0       1
5       7       10      0

0       0       1       0
3       1       1       2
3       0       2       2
3       0       1       3
```

根据路径矩阵 P,对于任何一对顶点 u、v,其路径可以从终点倒过来追踪到起点。即终点是 v,其前一个顶点是 P[u][v],再前一个顶点是 P[u][P[u][v]]……

输出任意两点 u 到 v 的最短距离路径的代码如下:

```
for (auto u = 0; u < n; u++)
    for (auto v = 0; v < n; v++) {
        if (u == v) continue;
        cout << u << "到" << v << "的逆向路径是:";
        cout << v << ',';            //终点
        auto w{ P[u][v] };
        for (auto w{ P[u][v] }; w != u;) {
            cout << w << ',';
```

```
            w = P[u][w];
        }
        cout << u << std::endl;
}
```

执行程序，输出结果：

```
0 到 1 的逆向路径是:1,0
0 到 2 的逆向路径是:2,1,0
0 到 3 的逆向路径是:3,0
1 到 0 的逆向路径是:0,3,2,1
1 到 2 的逆向路径是:2,1
1 到 3 的逆向路径是:3,2,1
2 到 0 的逆向路径是:0,3,2
2 到 1 的逆向路径是:1,0,3,2
2 到 3 的逆向路径是:3,2
3 到 0 的逆向路径是:0,3
3 到 1 的逆向路径是:1,0,3
3 到 2 的逆向路径是:2,1,0,3
```

5.7 习题

1. 下面程序的输出是什么？

```
#include<iostream>
using namespace std;
int main(){
    int x = 10;
    int& ref = x;
    ref = 20;
    cout << "x = " << x << endl ;
    x = 30;
    cout << "ref = " << ref << endl;
    return 0;
}
```

2. 关于下列程序，正确的是：
   A. 运行时错误　　B. 编译时错误　　C. 没问题　　D. 0

```
#include<iostream>
using namespace std;

int &fun(){
    int x = 10;
    return x;
}
```

```
int main(){
    fun() = 30;
    cout << fun();
    return 0;
}
```

3. 下面哪个定义是非法的？为什么？

(1) int ival = 1.01; (2) int &rvall = 1.01;

(3) int &rval2 = ival; (4) int &rval3;

4. 下列代码的输出是什么？

```
int i, &ri = i;
i = 5; ri = 10;
std::cout << i << " "<< ri << std::endl;
```

5. 请叙述下面这段代码的作用。

```
int i = 42;
int *pl = &i;
*pl = *pl * *pl;
```

6. 解释下列语句的含义，并说明错的语句的原因。

```
int i = 0;
```

A. double * dp = &i;

B. int *ip = i;

C. int *p = &i;

7. 下面这段代码中为什么 p 合法而 lp 非法？

```
int i = 42; void *p = &i; long *lp = &i;
```

8. 说明下述变量的类型和值。

(1) int * ip, *&r = ip;

(2) int i, *ip = 0;

(3) int * ip, ip2;

9. 下列哪些数组定义有错误？为什么？

```
unsigned buf_size = 1024;
```

(1) int ia[buf_size];

(2) int ib[4 * 7—24];

(3) int ic[4 * 7—28];

(4) char st[11] = "hello world";

10. 找出下列程序中的错误。

```
int v[] = { 1,2,3,4 };
v0 = 10.5;
int i = v[3] + 10;
v[2] = v[0] - i;
i = v[4];
```

11. 下面哪句是不合法的？原因是什么？

(1) const int buf;

(2) int cnt = 0;

(3) const int sz = cnt;

(4) ++cnt; ++sz;

12. 下面哪句是不合法的？原因是什么？

(1) int i = -1, &r = 0;

(2) const int i2=i, &r2 = i;

(3) const int i3=-i, &r3 = 0;

(4) const int * p1 = &i2;

(5) int * constp2 = &i2;

(6) const int * constp3 = &i2;

(7) const int & constr4;

13. 下面代码中哪些语句是错误的？为什么？

```
int a[5] = {};
int b[6] = {};
int* x = a;
int* const y = a;
b = x;
x = b;
y = b;
```

14. 下面代码中的变量 p1、ci、p2、p3、r 能否被修改？

```
int i{0};
int * const p1 = &i;
const int ci = 42;
const int *p2 = &ci;
const int * const p3 = p2;
const int &r = ci;
```

15. 在第 14 题的基础上，下面定义的变量有没有错误？为什么？

```
int *p = p3;
p2 = p3;
p2 = &i;
int &r1 = ci;
const int &r2 = i;
```

16. 下面哪句是不合法的？原因是什么？

(1) int i, * const cp;

(2) int *p1, * const p2;

(3) const int ic, &r = ic;

(4) const int * const p3;

(5) const int *p;

17. 对于第 16 题中的变量，下面哪句是不合法的？原因是什么？

(1) i = ic;

(2) p1 = p3;

(3) p1 = ⁣

(4) p3 = ⁣

(5) p2 = p1;

(6) ic = * p3;

18. 下面哪句是不合法的？原因是什么？

```
int i = 0;
int * const p1 = &i;
const int ci = 42;
const int *p2 = &ci;
const int * const p3 = p2;
const int &r = ci;
int *p = p3;
p2 = p3;
p2 = &i;
int &r = ci;
const int &r2 = i;
```

19. 说明下列哪些变量是 const 对象，哪些是指向或引用 const 对象的指针或引用。

```
const int v2 = 0;   int v1 = v2;
int *p1 = &v1, &r1 = v1;
const int *p2 = &v2, * const p3 = &v2, &r2 = v2;
```

20. 对于第 19 题中的变量，下列哪些语句是合法的？哪些是非法的？为什么？

```
r1 = v2;
p1 = p2;
p2 = p1;
p1 = p3;
p2 = p3;
```

21. 编写程序，从键盘输入 10 个整数到一个数组中，并将数组中的元素逆序放入另外一个数组中，最后输出这两个数组中的所有整数。

22. 从键盘输入 10 个整数到一个数组中，然后遍历数组求出这些整数中的最大值和最小值并输出。

23. 编写程序，输出一个乘法口诀表。

24. 将下列程序改成用 auto 和 rang for。

```
#include <iostream>
int main(){
    //声明一个 10×10 的数组
    const int numRows = 10;
    const int numCols = 10;
    int product[numRows][numCols] = { 0 };

    //计算乘法表
    for (int row = 0; row < numRows; ++row)
        for (int col = 0; col < numCols; ++col)
            product[row][col] = row * col;

    //打印表
    for (int row = 1; row < numRows; ++row)  {
        for (int col = 1; col < numCols; ++col)
            std::cout << product[row][col] << "\t";
        std::cout << '\n';
    }
    return 0;
}
```

25. 下面程序的输出是什么？

```
#include <iostream>
int main() {
    int i = 1;
    int const& a = i > 0 ? i : 0;
    i = 2;
    std::cout << i << a;
}
```

26. 关于下面代码，正确的输出是(　　)。

A. 3　　B. 0x822222232　　C. 语法错误　　D. 不确定

```
#include <iostream>
int main(int argc, char ** argv){
    //假设 x 的地址是 0x822222232
    int x = 3;
    int * & rpx = &x;
    std::cout << rpx << std::endl;
}
```

27. 解释下列程序。

```
const char * cp = ca;
while ( * cp) {
```

```
        cout << * cp << endl;
        ++cp;
    }
```

28. 从网上搜索表5-1中的C风格字符串处理函数的作用,并编写程序学习这些函数的使用。

29. 下面的函数Strlen()是模拟C风格字符串处理函数strlen(),请补充其代码,并运行程序测试这个函数。

```
int Strlen(const char * str){
    //编写你的代码
    //...
}
#include <iostream>
#include <cstring>
using namespace std;
int main(){
    char * s = "Hello world";
    std::cout <<"s 的长度是:"<< Stelen(s)<< endl;       //调用自己编写的 Strlen()函数
    std::cout <<"s 的长度是:"<< stelen(s)<< endl;       //调用标准库的 strlen()函数
}
```

30. 实现并测试简单选择排序算法。

31. 完善学生成绩分析程序,用动态内存分配存储所有学生的多门成绩,并根据每门成绩占总成绩的百分比计算总评成绩,分析期末成绩或总评成绩的平均分,分距(最高分减去最低分),不同分数区间(不及格、及格、中等、良好,优秀)的百分比等。要求:学生的成绩、每门成绩占总成绩的百分比都从键盘输入。

32. 用动态内存分配来表示二维矩阵,编写一个函数实现初始化一个矩阵、根据下标读写矩阵元素、两个矩阵相加或相乘、输出一个矩阵的功能。



第6章

函　数

6.1 函数是命名的程序块

6.1.1 最大公约数

两个正整数的最大公约数就是能被两者整除的最大正整数。给定 2 个正整数 m 和 n，假设用符号 GCD(m,n)表示两者的最大公约数，它们的最大公约数可以用下列式子来计算：

$$\text{GCD}(m,n)=\begin{cases} m & n=0 \\ \text{GCD}(n,m\%n) & n!=0 \end{cases}$$

其中，%表示求余数运算。因此，可以用如下迭代方法求两个正整数的最大公约数。

$$\begin{aligned}\text{GCD}(72,27) &= \text{GCD}(27,72\%27)=\text{GCD}(27,18)\\ &= \text{GCD}(18,27\%18)=\text{GCD}(18,9)\\ &= \text{GCD}(9,0)=9\end{aligned}$$

可以写出下面的求最大公约数的代码：

```
#include <iostream>
int main(){
    int m = 72,n = 27;
    while(n!= 0){
        int r = m % n;
        m = n; n = r;
    }
    std::cout <<"最大公约数是: "<< m << std::endl;
}
```

假如程序中接着又要求另外 2 个整数(如 36 和 24)的最大公约数，怎么办？只要重复使用上述代码(即复制、粘贴上述代码)就可以了。

```
#include <iostream>
int main(){
    int m = 72,n = 27;
    while(n!= 0){
        int r = m % n;
        m = n; n = r;
    }
    std::cout << m <<"和"<< n <<"的最大公约数是: "<< m << std::endl;

    m = 36; n = 24;
    while(n!= 0){
        int r = m % n;
        m = n; n = r;
    }
   std::cout << m <<"和"<< n <<"的最大公约数是: "<< m << std::endl;
}
```

假如这个程序其他地方或其他程序也要求最大公约数,可以重复这种代码的复制、粘贴过程,但这样的复制、粘贴会使程序代码越来越长。还有一个更严重的问题: 万一以后发现需要修改或改进求最大公约数的代码,就需要找出所有复制、粘贴的地方,去逐一替换或修改,这会变得非常麻烦。

解决这个问题的方法就是给求最大公约数的这段代码起一个名字,即定义所谓的**函数**,这段代码只要编写一次,然后通过这个**函数名**来调用这段代码而不需要复制、粘贴代码,称为**函数调用**。如果将来修改函数内部的代码,其他调用函数的地方不需要做任何修改。

这段求最大公约数的函数代码不应该只针对固定的 2 个整数而应该对任意 2 个整数都能求最大公约数,这就需要一种机制将 2 个整数参数化,并传递给这段代码。

看看具体做法:

```
//GCD 是函数名,圆括号内是参数化的 2 个整数 m,n,称为"形式参数",简称"形参"
//函数名前面的 void 关键字,说明这个函数不返回值
void GCD(int m,int n){
    //int m = 72,n = 27;
    while(n!= 0){
        int r = m % n;
        m = n; n = r;
    }
    std::cout << m <<"和"<< n <<"的最大公约数是: "<< m << std::endl;
}

int main(){
    int x = 72,y = 27;
    GCD(x,y);          //调用函数名叫 GCD 的代码,将 x,y 的值传递给
                       //被调用函数 GCD 的 2 个形参 m,n
    x = 36; y = 24;
    GCD(x,y);
}
```

在定义函数名为 GCD 的函数时，函数名 GCD 后的圆括号内就是这个函数可以接收的外部数据，称为函数的形式参数(简称形参)。在 main()函数中通过函数名 GCD()调用函数 GCD()时，会将变量 x、y 的值传递给(复制给)被调用函数 GCD()的 2 个形参 m、n，然后执行 GCD()函数中的代码。

因为 main()函数中通过语句 GCD(x,y)调用了函数 GCD()的代码，因此，称 main()函数为**调用函数**、GCD()函数为**被调用函数**。

求最大公约数的函数 GCD()的代码只要定义 1 次，就可以在其他函数如 main()函数中被多次调用。每次调用 GCD()函数时，调用函数将称为**实参**的变量 x、y 传递给被调用函数的 2 个形参 m、n，然后进入被调用函数 GCD()执行，被调用函数执行完后，就回到 main()函数中。

GCD()函数名前面的 void 关键字，说明这个函数不返回值或者说返回类型是 void(即**无类型**)。也可以让被调用函数返回一个非 void 类型(如 int 类型)的一个值(结果)，那么可以这样修改上述代码：

```
//GCD 是函数名,圆括号内是参数化的 2 个整数 m,n,称为"形式参数"
//函数名前面的 int,说明这个函数返回一个 int 类型的结果(值)
int GCD(int m,int n){
    while(n!= 0){
        int r = m % n;
        m = n; n = r;
    }
    return m;               //返回 int 类型的结果(值)m
}

int main(){
    int x = 72,y = 27;
    int gcd = GCD(x,y);    //用函数 GCD()的返回结果初始化 int 类型变量 gcd
    std::cout << m <<"和"<< n <<"的最大公约数是: "<< gcd << std::endl;

    x = 36; y = 24;
    gcd = GCD(x,y);
    std::cout << m <<"和"<< n <<"的最大公约数是: "<< gcd << std::endl;
}
```

实际上，第 3 章中求平方根和计算指数时已经使用过这种函数调用，如：

```
#include <cmath>          //需要包含声明 sqrt()和 pow()函数的头文件 cmath
int main(){
    double d = 6.8;
    std::cout << d <<"的平方根是: "<< sqrt(d)<< Std::endl;
    std::cout << d <<"的 3.4 次方: "<< pow(d,3.4)<< std::endl;
}
```

6.1.2 函数的定义

上面的代码定义了 2 个函数 main()和 GCD()。

1. 函数定义格式

函数的定义包括 4 个部分：返回类型、函数名、形参列表、函数体，其格式如下：

```
返回类型   函数名(形参列表)
{
    函数体(程序块)
}
```

返回类型说明这个函数返回结果的类型，如果返回类型是 void，说明不返回任何结果，否则，应返回这种类型或能自动转换成这种类型的结果(值)。

形参列表说明了这个函数将来被调用时可以接受哪些参数，每个参数的类型是什么。形参列表也可以是空的(表示不接受任何实际参数)。

函数体就是以{}包围的程序块。函数体通常包含多条语句，其中也可能有调用其他函数的**函数调用语句**(如 main()函数体中的“int gcd = GCD(x,y);”)。函数体也可以是空的，但空函数体的函数是没有用处的。

2. 形参

函数可以没有任何形参(如上面的 main()函数)，也可以有 1 个以上的形参，多个形参用逗号隔开，每个形参包含形参类型和形参名。同一个函数不能有同名的形参，函数内部定义的变量也不能和形参同名。如：

```
//可以是空的,但不能没有圆括号(),也可用 void 表示空的形参
void f1(){ /* ... */}
void f2(void){ /* ... */}

//形参可以用逗号隔开,但每个参数都必须说明其类型
void f3(int v1, v2){ /* ... */}          //错: v2 没有说明类型
void f4(int v1, int v2){ /* ... */}      //Ok

//2 个形参不能同名,形参也不能和函数内部的最外层局部变量同名
int f5(int v, int v){ /* ... */}         //错: 2 个形参不能同名
int f5(int v){
    int v;                               //错: 形参不能和函数内部的最外层局部变量同名
    //...
}
```

3. 返回类型

关于函数的返回值和返回类型，有下面的一些规定。

(1) 每个函数都必须说明其返回类型(后面将介绍的类的构造函数和类型转换函数

除外)。

(2) 大多数类型都可以用作返回类型,说明函数值的类型。返回类型也可以是 void,说明该函数不返回值。

(3) 可以用 auto 关键字,让编译器从函数的返回值自动推断函数的返回类型。

(4) 如果函数有多个 return 语句,这些 return 语句必须返回返回类型或能隐含转换为返回类型的值。

```
int g() { return 1; }                //通过 return 关键字,返回一个结果
                                     //结果的类型和返回类型一致或能转换成返回类型
double g1() { return 1; }            //通过 return 关键字,返回一个结果
                                     //结果的类型和返回类型一致或能转换成返回类型
void g2(){ std::cout <<"void 类型不需要返回值";}
void g3(){ return 1;}                //对于 void 返回类型,如果返回的是
                                     //非 void 类型的值,则编译出错
int g4(){ }                          //返回类型是 int 的必须返回一个可以转换成 int 的结果(值)
auto g5() {return 1; }               //通过 auto 推断返回类型是 int
auto g6() {std::cout <<"void 类型不需要返回值";}      //通过 auto 推断返回类型是 void
auto g7() {return void}              //return void 返回的类型是 void,即无类型
int g(){                             //一个函数可以有多个 return 语句,函数遇到 return 语句就
                                     //执行结束
    int i; std::cin >> i;
    if(i > 0) return 1;              //函数结束,返回 1
    else if(i < 0) return -1;        //函数结束,返回 -1
    else return 0;                   //函数结束,返回 0
}
```

(5) 不能返回非静态局部变量的指针或引用。例如:

```
//返回 int * 指针类型
int * fp(){
    int var;
    //...
    return &var;
}
//返回 int & 引用类型
int &fr(){
    int var;
    //...
    return var;
}
```

上述 2 个函数返回一个**非静态局部变量**(静态变量和非静态变量请看 6.2 节)的指针或引用,但**非静态局部变量**在函数结束后就不存在了,调用函数将来如果通过这个返回的指针或引用去访问这个不存在的变量,会导致灾难性的后果(程序崩溃)。

另外,不能从一个返回的初始化列表推断返回类型。下面的代码是错误的:

```
auto func(){ return {1,2,3}; }
```

6.2 静态变量

定义变量时，如果前面有 **static** 关键字，这个变量就称为**静态变量**。因此，一个程序块（包括函数）中的局部变量根据是否是静态变量可分为**静态局部变量**和**非静态局部变量**。前面接触的局部变量都是非静态局部变量。

先看下列程序：

```
int main(){
    while (true) {
        auto i{0};                  //i 是一个非静态局部变量
        if (i++ < 6) std::cout << i << '\t';
        else break;
    }
}
```

每次进入循环体时，创建一个变量 i（值为 0），然后因为满足 i<6，执行了 i++变为 1，并输出这个 1。当循环体执行完，回到 while(true)前，这个局部变量就销毁了，当下次再进入这个循环体，又会重新创建一个新的局部变量 i（值为 0），然后重复上述过程。因此，执行该程序，将进入无限循环，一直输出值 1。

如果将其中的声明“auto i{0};”修改为“static auto i{0};”。即将 i 定义为一个静态局部变量，执行该代码将输出结果：

```
1       2       3       4       5       6
```

程序正常结束。

这是因为静态变量一旦初始化，就会一直存在，不会因为执行完循环体开始下一个循环时而销毁。因此，当第 2 次进入循环体时，这个变量始终存在，且不会重新初始化，其值将是上一次循环体里的值，因此，这个值在每次循环时都在不断变化。当超过 6 时，就执行 break 语句退出整个循环，程序就正常结束了。

同样，一个函数中的变量当然也可以定义成静态局部变量。例如：

```
#include <iostream>
void  f() {
    static auto i{ 0 };             //i 是静态局部变量
    int j{ 0 };                     //j 是非静态局部变量
    i++;  j++;
    std::cout << i << '\t' << j << '\n';
}

int main() {
    f();
    f();
}
```

上述的 f()函数中定义了一个静态局部变量 i 和一个非静态局部变量 j，在 main()函数中调用 2 次 f()的时候，静态变量 i 始终存在，而非静态变量每次都重新创建和销毁。因此，程序输出是：

```
1    1
2    1
```

因此，静态变量一旦初始化，就会一直存在，直到程序结束才销毁。这点和全局变量类似，但全局变量是在程序开始执行时就初始化，且能被程序的所有代码访问，而静态局部变量的作用域只存在于其定义的程序块，外部无法访问它。

6.3 函数的形参

6.3.1 参数传递

调用一个函数时，其形参被创建并用实参初始化。形参初始化和变量初始化是一样的。函数的形参分为：引用形参和非引用形参（也称值形参）。

- 当形参是引用形参时，形参被绑定到实参（即形参是实参的别名）。
- 当形参不是引用形参时，实参的值被复制（拷贝）给形参。

例如：

```
#include <iostream>
int f(int var, int &ref){
    var++;
    ref++;
    std::cout << var <<'\t'<< ref << std::endl;
}
int main(){
    auto x = 1, y = 1;
    f(x, y);
    std::cout << x <<'\t'<< y << std::endl;
}
```

在 main()函数执行 f(x，y)即调用函数 f()时，main()函数的变量 x 被赋值给 f()函数的形参 var，而 f()函数的引用形参 ref 则是 main()函数的变量 y 的引用（别名），即 f()函数的 ref 和 main()函数的 y 是同一块内存的不同名字，如图 6-1 所示。

f()函数执行 ref++实际就是对 ref 和 y 命名的同一块内存的内容执行自增运算。因此，这块内存的值（即 y 的值）就是 2。f()函数对 var 的自增，虽然确实使得 var 对应的内存值变为 2，然而 main()函数的 x 对应的内存块没有受到任何影响，因为 var 和 x 是各自独立的两块内存。当 f()函数执行完后，其局部变量（包括形参）就销毁了（不存在了）。程序又回到刚

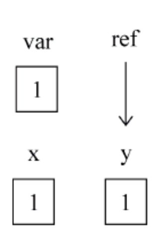


图 6-1 值形参和引用形参

才函数调用 f(x,y)的下一条语句即输出语句处继续执行。因此，运行程序的输出结果为：

```
1 2
```

每个程序有一个自己的堆栈区，用以维护函数之间的调用关系。栈是一种先进后出的数据结构，如同厨房里的一堆盘子，后放的盘子总是放在最上面(称为**栈顶**)，盘子也是从栈顶被拿走。将盘子放入栈顶称为**入栈**，而从栈顶拿走盘子称为**出栈**。

如图 6-2 所示，当执行 main()函数还没调用 f()函数时，程序的栈顶是 main()函数的局部变量，当开始执行 f(x,y)时，f()函数的局部变量将进入栈顶(入栈，即这些局部变量被创建了)，表示当前执行的函数是 f()函数，执行完 f()函数，程序栈顶的 f()函数的局部变量被弹出栈(即这些局部变量被销毁了)，栈顶又是 main()函数的局部变量，表示又回到了 main()函数。这个时候 main()函数将从调用 f(x,y)的下一条语句继续执行。

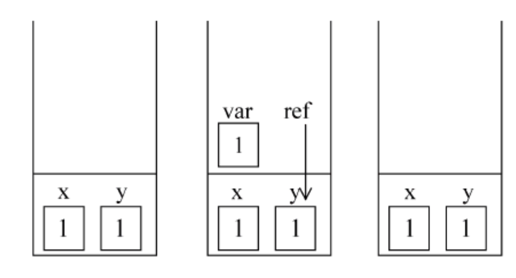


图 6-2 程序堆栈：维护函数调用关系

6.3.2 默认参数

函数的形式参数可以有默认值。例如：

```
int Pow(int x, int e = 2) {
    auto ret{1};
    for (auto i = 0; i < e; i++)
        ret *= x;
    return ret;
}

#include <iostream>
int main() {
    std::cout << Pow(3) << '\t'<< Pow(3, 4) <<'\n';
}
```

其中，Pow()函数的第 2 个形参 e 有一个默认值，在调用这个函数时，如果没有提供对这个形参初始化的实参，则形参的值就是默认值。因此，函数调用 Pow(3)中形参 e 的值是 2，而 Pow(3,4)中形参 e 的值是 4。执行该程序的结果如下：

```
9       81
```

注意：定义函数时，有默认值的形参总是在非默认形参的后面，如果将 Pow()函数写成如下形式：

```
int Pow(int e = 2, int x)
```

则编译器会报告错误。

6.3.3 数组作为形参

可以将函数的形参写成数组的样子，除了这个数组形参外，通常还必须有另外的形参说明这个数组的大小。例如：

```
#include <iostream>
void PrintArr(int arr[], int n) {    //n说明形参,arr表示数组的大小
    for (auto i = 0;i<n;i++)
        std::cout << arr[i] << '\t';
}
int main() {
    int a[]{ 7,2,4,19 };
    PrintArr(a, 4);
}
```

当形参写成数组的形式时，这个形参实际上并不是一个真正的数组，而是一个指向数组的指针变量，编译器实际上将上述函数转换成如下形式的形参：

```
void PrintArr(int *arr,int n)
```

即这个 arr 形参实际就是一个 int * 类型的指针变量，而并不是一个真正的数组。因此，不能对 arr 用 range for 循环去访问其中的数组元素。例如：

```
void PrintArr(int arr[], int n) {
    for (auto e :arr)
        std::cout << e << '\t';
}
```

将产生编译错误：

```
error C3312: 未找到可调用的"begin"函数(针对类型"int []")
…
```

当然，可以通过指针去遍历数组中的元素。例如：

```
void PrintArr(int arr[], int n) {
    for (auto p = arr;p!= arr + n;p++)
        std::cout << *p << '\t';
}
```

因此，在数组形参中的[]里指定数组大小是没有任何意义的。下列都是等价的，最终都转换为最下面的指针形参的形式：

```
void PrintArr(int arr[], int n);
void PrintArr(int arr[2], int n);
```

```
void PrintArr(int arr[10], int n);
void PrintArr(int * arr, int n);    //上面 3 种形式最终都会转换为这个形式
```

但如果形参是一个引用数组的形参，那么这个形参就引用了实参那个数组，这个时候形参引用的就是一个真正的数组而不是一个指针了，此时的形参必须说明引用数组的大小，也就不必另外传递一个大小形参了。因为这个形参就是数组，在函数里，也可以用 range for 去访问数组的元素。例如：

```
#include <iostream>
void SquareArr(int(&arr)[4]) {       //arr 引用的是 int[4]类型的数组,即引用的是有 4 个 int 元
                                     //素的数组
    for (auto &e : arr)              //arr 既然是一个真正数组,就可以用 range for 循环
        e *= e;
}
int main() {
    int a[]{ 7,2,4,19 };
    SquareArr(a);
    for (auto e : a)
        std::cout << e << '\t';
}
```

执行程序，输出结果：

```
49      4       16      361
```

和定义多维数组一样，也可以定义一个多维数组的形参。如果这个形参不是引用形参，同样实际传递的是一个指针，因此，和一维数组形参一样，说明数组形参的最低维（即最外层）的大小是不需要也是没有任意意义的，但必须指明其他每一维的大小。

```
void f(int arr[][4], int n);         //等价于 void f(int ( * arr)[4], int n);
void g(int brr[][4][5], int n);      //等价于 void g(int ( * brr)[4][5], int n);
```

arr 和 brr 实际是一个指针变量，如 arr 的类型是 int (*)[4]，也即它是一个指向数组类型 int[4]的指针变量，即它指向的是一个 4 个 int 类型数构成的**数组类型**的变量而不是指向一个 int 类型的变量，也不是指向其他如 int[8]数组类型的对象，即它指向的变量类型必须是 int[4]的。对 arr 如果执行 arr++，则 arr 指向变量将跳过 int[4]数组的 4 个整数，即偏移 4 个整数占据的大小（如 16 字节）而指向下一个 int[4]数组，即二维数组的第 2 行。可以用如下代码来验证这一点：

```
#include <iostream>
void h(int arr[][4], int n) {
     //p 指向一个 int[4]数组的指针,即指向一行对应的那个数组
    //p++就指向下一个 int[4]数组
     for (auto p = arr; p != arr + n; p++) {
          for (auto e : *p)          //*p 就是数组 int[4],所以可以用 range for
```

```
            std::cout << e << '\t';
        std::cout <<'\n';
    }
}
int main() {
    int a[][4]{ {1,2,3,4},{5,6,7,8},{9,10,11,12} };
    h(a,3);                    //必须传递 a 的大小
}
```

执行程序,输出结果:

```
1        2        3        4
5        6        7        8
9        10       11       12
```

6.3.4 const 与形参

函数的形参作为函数的局部变量,当然可以用 const 修饰。例如:

```
void f(const int x, const int y);
void g(const int *p, const int n);
void h(int * const q, const int n);
```

函数 f()的 2 个形参都是 const int 类型,即在函数 f()中,它们是不能被修改的 int 变量。函数 g()中的 p 是指向 const int 类型变量的指针,即在函数 f()中,不能通过 p 修改它指向的 const int 变量,但 p 本身是可以被修改的。h()函数的 q 是 const 变量,即在函数 h()中,q 是不可以被修改的,但它指向的 int 类型变量是可以被修改的。

如果不希望函数修改形参,可以像 x、y、n、q 那样,将它们定义为 const 对象(变量)。如果想禁止函数修改指针形参指向的变量,可以将指针变量指向的量定义为 const。如果既不允许修改指针形参本身,也不允许修改它指向的那个变量,则可以如下面的 s 这样定义:

```
void f2(const int * const s, const int n);
```

因为普通的数组形参就是指针形参,数组形参和 const 的结合是类似的。同样。const 修饰引用形参也是类似的。因为 const 和形参的结合,就是 const 和变量的结合(可以回顾 5.5 节)。

6.3.5 可变数目的形参

有时,无法提前预知给一个函数传递的参数个数,如编写一个函数求一个学生的平均分数,但不知道实际运行中到底有几门课程。虽然可以通过数组指针和数组大小来实现这一目的,但假如希望传递的都是课程分数而不包含课程的数目,这就需要定义一个能接受可变数目参数的函数。

C++从 C 语言中继承了一个 3 个点(…)的可变形参。但 C++还有更好的方法,一种方

法是用 C++标准库的 std::initializer_list < T >类型定义函数的形参。注意这个 initializer_list 是一个所谓的类模板(第 10 章会介绍模板),需要通过在箭头<>之间的一个具体数据类型 T,构造一个完整的数据类型 std::initializer_list < T >。

例如:

```
#include <iostream>
//scores 是 std::initializer_list<double>类型的变量
double average(std::initializer_list<double> scores){
    auto n{ 0 };
    double all{ 0 };
    for (auto score : scores) {
        all += score; n++;
    }
    if(n>0)    return all /= n;
    return 0;
}
```

这个 average()函数的形参 scores 是 double 类型的 std::initializer_list < double >类型的变量。表示可以给这个形参传递的是数目变化的多个 double 类型实参,传递的实参会被打包进这个 scores 对象中,可以通过 rang for 循环访问里面的每个 double 值。average()函数通过这个循环计算总分、统计个数,然后计算出平均值。

下面的 main()函数多次调用 average()函数,传递不同数目的 double 实参:

```
int main() {
    std::cout << average({   }) << '\n';
    std::cout << average({60.}) << '\n';
    std::cout << average({ 50.,80 }) << '\n';
    std::cout << average({ 90,50.,80 }) << '\n';
}
```

执行程序,输出结果:

```
0
60
65
73.3333
```

需要注意的是,std::initializer_list < T >类型的形参在函数中是 const 对象,是不可以被修改的。

另外,传递给该形参的必须都是同一个 T 类型的实参,不能传递不同类型的实参。C++可以通过**可变模板参数**(见第 10 章)实现传递不同类型的可变数目的实参。

要使用 std::initializer_list 模板,需要包含头文件< initializer_list >,但< iostream >已经包含了该头文件。

6.4 递归函数：调用自身的函数

6.4.1 递归和递归函数

递归是一个任务分解的解决问题的方法，一个大的问题如果能够分解成和它类似的子问题，且子问题的解决方法和大问题是一样的，只不过问题的规模有所区别而已，那么这种情况下就可以采用递归的方法来解决这个问题。

如求一个"n 的阶乘"问题，它可以通过 n 和"$(n-1)$的阶乘"相乘而得到，也就是规模为"n 的阶乘"的问题分解成了规模更小的"$(n-1)$的阶乘"问题。即：$n!=n*(n-1)!$

假如 n 的阶乘的求解过程用一个函数 fact(n)描述，可以编写下面的代码来求 n 的阶乘。

```
#include <iostream>
int fact(int n) {
    if (n == 1)                    //如果 n 等于 1,就直接返回值 1
        return 1;
    return n * fact(n - 1);        //fact(n)等于 n 和 fact(n-1)的乘积
}
```

可以看到，函数 fact(n)的内部代码存在调用该函数自身的函数调用语句"return n * fact(n—1);"。这种函数在其内部存在调用该函数自身的语句，就称为**递归函数**。

当 $n=1$ 时，n 的阶乘问题就不需要再分解了，即不需要再递归为更小的子问题了。这种不需要再分解的问题，称为**基问题**或**基情形**。

因此，编写递归函数，一定要根据是否是基情形而分别处理。

下面的 main()主函数调用函数 fact(4)计算 4 的阶乘。

```
int main() {
    std::cout << fact(4) << '\n';    //输出：24
}
```

执行程序，输出结果：

```
24
```

递归是一个嵌套的过程，如 fact(4)的递归计算过程如下：

```
fact(4)
4 * fact(3)
4 * (3 * fact(2))
4 * (3 * (2 * fact(1)))
4 * (3 * (2 * 1))
4 * (3 * 2)
```

```
4 * 6
24
```

1. 斐波那契数列

斐波那契数列又称黄金分割数列，因数学家列昂纳多·斐波那契(Leonardoda Fibonacci)以兔子繁殖为例子而引入，故又称为“兔子数列”，指的是这样一种数列$\{f(n)|n=0,1,2,\cdots\}$：

$$\begin{cases} f(0)=1 \\ f(1)=1 \\ f(n)=f(n-1)+f(n-2) \quad n\geqslant 2 \end{cases}$$

即繁殖到第n代的兔子总数是第$n-1$代的兔子总数和第$n-2$代的兔子总数之和。

可编写如下的递归函数求$f(n)$：

```
#include <iostream>
int fib(int n) {
    if (n <= 2)                    //基情形
        return 1;
    else                           //递归情形
        return fib(n - 1) + fib(n - 2);
}
int main() {
    for (int i{1};i!= 8;i++)
        std::cout << fib(i) << '\t';
}
```

执行程序，输出结果：

```
1       1       2       3       5       8       13
```

2. 最大公约数

前面的最大公约数也是一个递归问题。即当$n>0$时，m、n的最大公约数就是n和$m\%n$的最大公约数；而$n=0$时为基情形，公约数就是m。

$$\gcd(m,n)=\begin{cases} m & n=0 \\ \gcd(n,m\%n) & n!=0 \end{cases}$$

可写出如下的递归函数：

```
#include <iostream>
int gcd(int m, int n) {
    if (n == 0)
        return m;
    else
        return gcd(n, m % n);
}
```

```
int main() {
    std::cout << gcd(72,27) << '\t'<< gcd(24, 36) << '\t';
}
```

执行程序,输出结果:

```
19        12
```

6.4.2 实战:二分查找的递归实现

5.6.1 节的二分查找问题,可以看成一个递归问题:在非空的原序列上的查找问题,被分解为 3 个子问题。

(1) 和中间的元素的直接比较问题。

(2) 左区间上的查找问题。

(3) 右区间上的查找问题。

而左、右子区间的二分查找和原区间的二分查找过程是一样的。因此,可以写出基于递归的二分查找程序。

```
#include <iostream>
//在 a[L,H]上查找值 Value
auto binarySearch(int a[], const int L, const int H, int value) {
    if (L > H)                                              //空序列
        return -1;
    auto Middle = (L + H) / 2;
    if (a[Middle] == value)                                 //(1)中间元素直接比较
        return Middle;
    else if (value < a[Middle])
        return binarySearch(a, L, Middle - 1, value);      //(2) 左区间查找
    else
        return binarySearch(a, Middle + 1, H, value);      //(3) 左区间查找
}
int main() {
    int arr[]{ 5, 7, 12, 25, 34, 37, 43, 46, 58, 80, 82, 105 };
    std::cout << binarySearch(arr, 0, 11, 25) << '\t';
    std::cout << binarySearch(arr, 0, 11, 13) << '\n';
}
```

运行程序,输出结果:

```
3        -1
```

6.4.3 实战:汉诺塔问题

汉诺塔是由法国数学家爱德华·卢卡斯在 1883 年发明的(他的灵感来自一个印度教传说):假设有 A、B、C 3 个柱子,其中 A 柱子上有 $N(N>1)$个盘子,盘的尺寸从下到上依次变小。现在要求将盘子全部移到 C 塔,每次只能移动一个盘子,且小盘必须在大盘之上,当

然，盘子只能放在这 3 个柱子之一上，如图 6-3 所示。

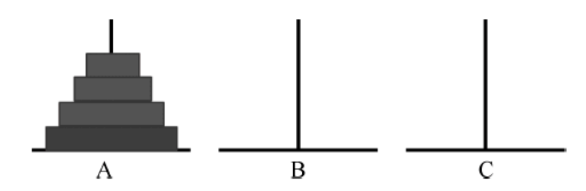


图 6-3 汉诺塔问题

假如采用蛮力尝试法，处于一个柱子上的盘子最多有 2 个选择(移动到另外 2 个柱子之一)，N 个盘子都这样尝试，至少需要 N 次尝试，因此至少需要移动2^N-1 次，但由于每个盘子不可能只尝试一次，所以总的次数会远远大于这个数字。

如果采用“分而治之”的解决问题方法，可以将“N 个盘子的移动(从 A 柱借助于 B 柱移到 C 柱)”分解为如下 3 个子问题(见图 6-4)。

- (上面的)$N-1$ 个盘子的移动(从 A 柱借助于 C 柱移到 B 柱)。
- 最大盘子的移动：直接从 A 柱移到 C 柱。
- $N-1$ 个盘子的移动(从 B 柱借助于 A 柱移到 C 柱)。

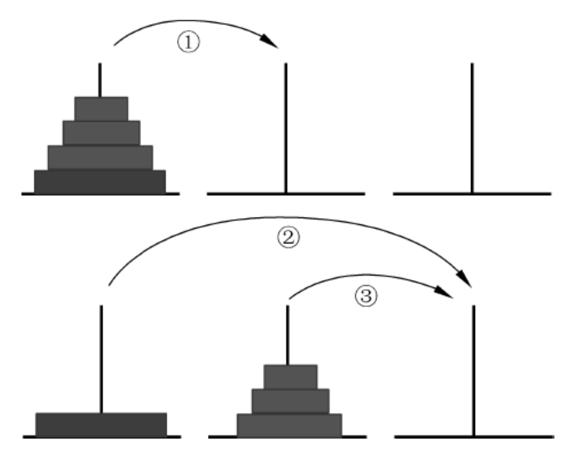


图 6-4 汉诺塔的递归分解

当然，对于 $N<1$ 的基问题，不需要移动。因此，可以写出下列代码：

```
#include <iostream>
//一个盘子：直接移动
void moveDisk(int i, const char x, const char y) {
        std::cout << "moving disk" << i << " from " << x << " to " << y << '\n';
}

//参数：盘数，起始柱，中转柱，目标柱
void move(int n, const char a, const char b, const char c) {
    if (n < 1) return;
    move(n - 1, a, c, b);            //n-1个盘子从A柱借助于C柱移到B柱
    moveDisk(n, a, c);
    move(n - 1, b, a, c);            //n-1个盘子从B柱借助于A柱移到C柱
}
int main() {
```

```
    move(3, 'A', 'B', 'C');
}
```

执行程序，输出结果：

```
moving disk 1   from A to C
moving disk 2   from A to B
moving disk 1   from C to B
moving disk 3   from A to C
moving disk 1   from B to A
moving disk 2   from B to C
moving disk 1   from A to C
```

6.4.4 实战：快速排序算法

对一组数进行排序的快速排序算法是一个递归过程：首先在这组数中随机取一个作为基准，将这组数分为2部分，其中一部分的所有数不超过基准，而另外一部分的所有数不小于基准。如有一组数：

34，2，89，47，29，13

假设任意取一个数（通常习惯取第1个）34作为基准，然后将这组数按照34分为2部分：

2，29，13，[**34**]，89，47

上述过程称为**"一次划分"**。对划分好的2部分重复上述过程，如此进行下去，直到每个部分的长度不超过1。

可以写出如下代码：

```
using T = int;                          //T是数据元素的类型
//qsort()是对[start, end]区间的元素进行快速排序过程
void qsort(T arr[], const int start, const int  end) {
    if (start >= end) return;
    //partition将[start, end]的序列一次划分为2部分,返回的pivot是基准的位置
    auto pivot = partition(arr, start, end);      //先对原序列一次划分
    qsort(arr, start, pivot - 1);   //对[start, pivot - 1]的序列调用qsort()快速排序过程
    qsort(arr, pivot + 1, end);     //对[pivot + 1, end]的序列调用qsort()快速排序过程
}
```

其中，qsort()是对一个区间[start，end]进行快速排序的递归过程，如果是一个合法的区间，该过程主要分为3步：先用partition()函数将区间"一分为二"并返回基准元素的位置，然后对基准元素的左右2部分区间重复这个qsort()快速排序过程。

如何对一个区间"一次划分"？如图6-5所示，可以使用首尾2个指示器，当右指示器指向的元素大于或等于基准元素时，该指示器向左移动，否则就停止；当左指示器指向的元素小于或等于基准元素时，该指示器向右移动，否则就停止。当2个指示器都停止时，交换它们指向的元素的值，就可以继续"2个指示器向内靠拢"的过程。当右指示器位于左指示器左边时，此时，左指示器的位置就是基准的位置，将该位置元素和基准交换就完成了一次划分。

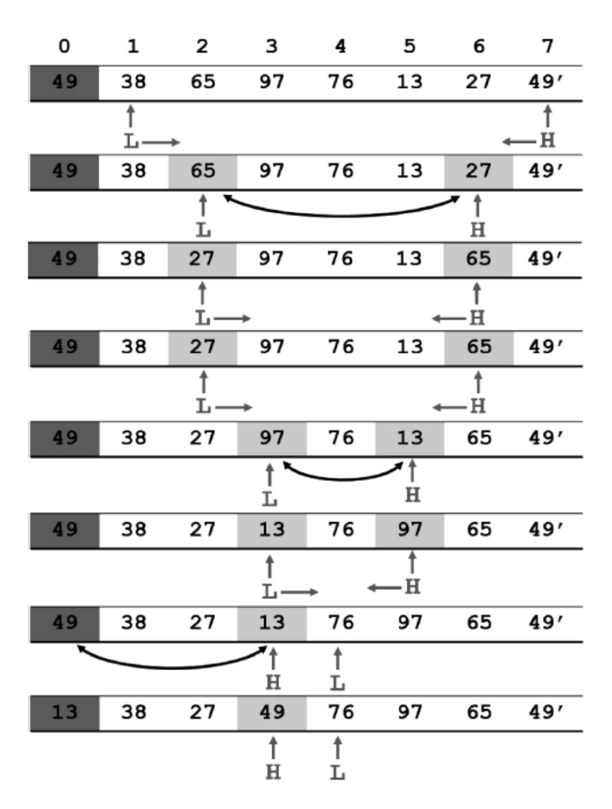


图 6-5 “一次划分”过程

对一个区间完成一次划分的函数 partition()的代码如下：

```
int partition(T arr[], const int start, const int end){
    auto pivotvalue{ arr[start] };
    auto L = start + 1, H = end;

    auto done{ false };
    while (!done) {
        while (L <= H and arr[L] <= pivotvalue)
            L = L + 1;
        while (arr[H] >= pivotvalue and H >= L)
            H = H - 1;
        if (H < L)
            done = true;
        else {
            //交换 L,H 的值
            auto temp{ arr[L] };
            arr[L] = arr[H];
            arr[H] = temp;
        }
    }
    //交换 L 和 start 的值,将基准元素移到基准位置 L
    auto temp{ arr[L] };
    arr[L] = arr[start];
    arr[start] = temp;
```

```
    return L;
}
```

调用对一个区间的快速排序递归函数 qsort(),可对一个序列进行快速排序:

```
void quickSort(T arr[], const int n) { //对 n 个元素的数组 arr 的快速排序
      qsort(arr, 0, n - 1);           //调用对一个区间快速排序过程 qsort()
}
int main() {
    int a[]{ 49, 38, 27, 97, 76, 13, 27, 49 };
    quickSort(a, 8);
    for (auto i{ 0 }; i != 8; i++)
          std::cout << a[i] << '\t';
    std::cout << '\n';
}
```

执行程序,输出结果:

```
13      27      27      38      49      49      76      97
```

6.4.5 实战:迷宫问题

给出一个迷宫,指明起点和终点,找出从起点出发到终点结束的有效可行路径,就是迷宫问题(maze problem)。图 6-6 所示是一个迷宫。

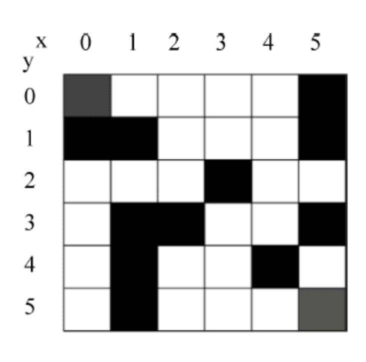


图 6-6 一个迷宫

迷宫可以用二维数组来表示。0 表示通路,1 表示障碍,2 表示终点。坐标以行和列表示,均从 0 开始,给定起点(0,0)和终点(5,5),迷宫表示如下:

```
maze = [[0, 0, 0, 0, 0, 1],
        [1, 1, 0, 0, 0, 1],
        [0, 0, 0, 1, 0, 0],
        [0, 1, 1, 0, 0, 1],
        [0, 1, 0, 0, 1, 0],
        [0, 1, 0, 0, 0, 2]]
```

迷宫求解问题可以描述为一个递归过程:对于一个当前位置,判断该位置是否是终点(2)、墙(1)、已经走过(3),如果不是上述情况,说明该位置可通但未走过(0),可从该位置走向其 4 邻(上、下、左、右 4 个位置),对新位置重复这个过程。

```
bool  go_maze(int maze[][6], const int x, const int y, const int n = 6) {
      //该位置(x,y)是否是终点(2)、墙(1)、已经走过(3)
      if (maze[x][y] == 2) {
```

```
            std::cout << "到达终点：" << x << "," << y << '\n';
            return true;
        }
        else if (maze[x][y] == 1) {
            //std::cout << "墙：" << x << "," << y<<'\n';
            return false;
        }
        else if (maze[x][y] >= 3) {
            //std::cout << "已经访问过：" << x << "," << y << '\n';
            return false;
        }
        //从该位置向 4 邻探索
        std::cout << "访问：" << x << "," << y << '\n';
        maze[x][y] = 3;                //标记该位置已经访问过
        //向 4 邻探索
        if ((x < n - 1 and go_maze(maze,x + 1, y))
            or (y > 0 and go_maze(maze, x, y - 1))
            or (x > 0 and go_maze(maze, x - 1, y))
            or (y < n - 1 and go_maze(maze, x, y + 1)))
            return true;
        maze[x][y] = 4;                //此位置不通
        return false;
}
```

下面的函数 print()用于输出迷宫。

```
void print(int maze[][6], const int n = 6) {
    for (auto i = 0; i < n; i++) {
        for (auto j = 0; j < n; j++) {
            std::cout << maze[i][j] << ' ';
        }
        std::cout << "\n";
    }
}
```

在主函数里调用上述走迷宫函数。

```
int main() {
    int maze[][6] = { {0, 0, 0, 0, 0, 1},
        {1, 1, 0, 0, 0, 1},
        {0, 0, 0, 1, 0, 0 },
        {0, 1, 1, 0, 0, 1},
        {0, 1, 0, 0, 1, 0 },
        {0, 1, 0, 0, 0, 2} };
    go_maze(maze, 0, 0);
    print(maze);
}
```

运行程序，输出结果：

```
访问：0,0
访问：0,1
访问：0,2
访问：1,2
访问：2,2
访问：2,1
访问：2,0
访问：3,0
访问：4,0
访问：5,0
访问：1,3
访问：0,3
访问：0,4
访问：1,4
访问：2,4
访问：3,4
访问：3,3
访问：4,3
访问：5,3
访问：5,2
访问：4,2
访问：5,4
到达终点：5,5
3 3 3 3 3 1
1 1 3 3 3 1
4 4 4 1 3 0
4 1 1 3 3 1
4 1 4 3 1 0
4 1 4 3 3 2
```

思考：打印的位置还包括回退的位置，如果打印一个没有回退的路径？

6.5 函数重载与重载解析

6.5.1 函数重载

C++中同一个作用域中可以定义多个同名的函数，只要它们的形参列表不同。定义多个同名的不同函数称为**函数重载**。下面的 4 个函数 f()都具有不同的形参列表，在 C++中是完全合法的。

```
int f() {/ * ... * /}
int f(int) {/ * ... * / }
int f(int,int) {/ * ... * / }
double f(double) {/ * ... * / }
```

函数名及其形参列表构成了函数的签名，即只要函数的签名不同，就是不同的函数。同

名函数表示这些函数具有类似的行为，只是其参数不同而已。和用不同名字命名这些函数相比，提高了代码的可读性。

尽管可以定义多个同名的不同函数，但不允许出现 2 个以上函数签名相同而返回类型不同的函数，因为这些都属于同一个函数，属于**"重定义"**。C++不允许多次重定义一个变量或函数，即一个函数或变量只能定义 1 次。

下面的 2 个函数 f()属于重定义，是不允许的：

```
int f(int) {/ * ... * / }
double f(int) {/ * ... * / }
```

何为"形参不同"？形参不同是指形参的个数不同或对应参数的类型不同，形参不同和形参名无关。如：

```
int f(int a)
int f(int b)
```

2 个函数的形参是完全一样的，都是 int 类型，这属于"函数重定义"，是不允许的。

形参不同和形参是否是 const 也无关。下面的 2 个函数 f()的签名是一样的，两个 g()函数签名也是一样的，都属于重定义。

```
int f(int)
int f(const int)

int g(int * )                    //int 对象的指针
int g(int  * const)              //int 对象的常指针
```

但下面的 2 个同名函数 f()(或 g())的形参是不同的：

```
int f(int&)                      //int 对象的引用
int f(const int&)                //const int 对象的引用

int g(int * )                    //int 对象的指针
int g(const int  * )             //const int 对象的指针,指针不是 const
```

通常，如果函数不会修改形参，应将形参设置为 const，就可以接受 const 对象或 non-const 对象的实参，即采用 f(const int)而不是 f(int)的形式。如果写成 f(int)形式，就不能接受 const int 的实参(包括文字量)。对于指针或引用也是一样的，如果不会修改指针型形参，应将该指针设置为 const 指针，即 g(int * const p)。如果不会修改指针指向或引用绑定的对象，应该将指针或引用设置为 const 对象的指针或引用，即 g(const int *)或 f(const int&)的形式。

6.5.2 重载解析

函数调用时，如果有多个同名的函数，编译器根据实参来选择一个最合适的函数。这个选择最佳函数的过程称为**重载解析**。

重载解析是根据实参和形参的匹配情况来选择最佳匹配函数。参数匹配过程会涉及类型转换。

重载解析的过程如下。

- 确定候选匹配函数集：同名的、可见的、数目匹配、可类型转换的同名函数。
- 按照实参和形参匹配度来选择最佳的匹配函数。匹配度分为精确匹配、提升匹配、标准转换匹配、自定义转换匹配。

例如：

```
void f()
void f(int)
void f(int,int)
void f(double,double = 3.14)
```

对于函数调用 f(5.6)，参数数目匹配的只有 f(int)和 f(double，double=3.14)。但后者不需要类型转换，是精确匹配，因此最佳匹配只有后者，所以匹配成功。而对于 f(3，5.6)，候选匹配函数值 f(int，int)和 f(double，double=3.14)两者都可以通过标准类型转换匹配，即有 2 个同样匹配度的函数，因此匹配失败。

因此，重载解析的结果有 3 种。

- 只有唯一的最佳匹配函数。匹配成功。
- 没有找到任何匹配函数。匹配失败。
- 找到多个可以匹配的函数，无法区分谁是最佳匹配。匹配失败。

对于有多个最佳匹配的，可以在函数调用时通过强制类型转换选择唯一的最佳匹配函数，例如可以使用 f(3，static_cast < int >(5.6))将第 2 个参数转换为 int 类型，这时 f(int，int)就是唯一的最佳匹配函数，匹配成功。

实参和形参匹配按照优先次序分为如下 4 种类型。

- 精确匹配：无须任何类型转换或只做平凡转换（如数组名到指针、函数名到函数指针、T 到 const T）。
- 提升匹配：整数提升（小整型总是转换为 int 或更大的整型）；浮点提升（小浮点型总是转换为更大的浮点型）。如：char、unsigned char、bool、short 到 int，float 到 double。如：

```
void ff(int) { std::cout << "f(int)"; }
void ff(short) { std::cout << "f(short)"; }
int main() {
    ff('c');                        //char 提升为 int,因此调用的是 ff(int)
}
```

- 标准转换匹配：除提升匹配之外的任何算术类型之间的相互转换（如 int 和 double 之间的相互转换、int 到 unsigned int 的转换），枚举类型到任何算术类型的转换，派生类指针到基类指针的转换，类型指针到无类型指针（T * 到 void * ）的转换。
- 自定义转换匹配：如类（见第 7 章）的构造函数或类型转换运算符定义的类型转换。

6.5.3 const 对象的引用或指针

在重载解析时，实参对形参的初始化和普通变量的初始化是一样的。例如，可以用 const 或 non-const 对象的指针或引用去初始化 const 对象的指针或引用，反过来，不能用 const 对象的指针或引用去初始化 non-const 指针或引用。

先看下列代码回顾一下普通变量的指针或引用的初始化：

```
const int ci = 3;       //const 对象可以用 const 对象初始化
int i = ci;             //non - const 对象可以用 const 对象初始化
const int j = i;        //const 对象可以用 non - const 对象初始化
const int &cri = i;     //const 对象的引用可用 const 或 non - const 值初始化,包括文字量
const int &cr3 = 3;     //const 对象的引用可用 const 或 non - const 值初始化,包括文字量
const int &crj = j;     //const 对象的引用可用 const 或 non - const 值初始化,包括文字量
int &r = ci;            //non - const 对象的引用(普通引用)不能用 const 对象初始化
int &r = 3;             //non - const 对象的引用(普通引用)不能用 const 对象初始化,包括文字量

const int * cp = &i;  //ok: const 对象的指针可用 const 或 non - const 的指针(地址)初始化
const int * cpj = &j; //ok: const 对象的指针可用 const 或 non - const 的指针(地址)初始化
const int * cp3 = &3; //错: 文字量没有地址
int *p = &i;            //普通指针(non - const 对象的指针)可用 non - const 的指针(地址)初始化
int *p2 = cp;           //error: 普通指针不能用 const 对象的指针初始化: p2 和 cp 类型不匹配
int *pj = &j;           //error: 普通指针不能用 const 对象的指针初始化: pj 和 &j 类型不匹配
```

对于函数，涉及 const 的指针或引用的形参的初始化也是一样的，如：

```
void fun(int * ) {/ * ... * / }
void fun(int &) {/ * ... * / }
void g(const int &) {/ * ... * / }

int main() {
    int i = 0;
    const int ci = i;
    unsigned  ui = 0;
    fun(&i);            //调用 fun(int * )
    fun(&ci);           //错: 不能将 const int 的指针转换为 int *
    fun(i);             //调用 fun(int &)
    fun(ci);            //错: 不能将普通引用 int & 绑定到一个 const 对象 ci
    fun(18);            //错: 不能将普通引用 int & 绑定到一个文字量
    fun(ui);            //错: 类型不匹配,ui 是 unsigned
    g(37);              //ok: cosnt int 的引用可以用文字量初始化
}
```

6.6 inline 函数

对于一个代码很少的函数，函数调用时传递参数和得到返回结果的开销可能比函数体内部代码的开销还大，并且编译器生成用于参数传递或返回结果的代码可能比函数体代码

占用更多内存，对于这种短小的函数，可以在函数定义前用关键字 **inline** 声明为内联函数，指示编译器在编译时将函数调用语句替换为函数体的代码并对函数体的局部变量名做一些调整。从而可避免函数调用的开销，提高程序效率，且使程序代码更短。如下面的 add()函数前通过添加关键字 inline 被声明为**内联函数**。

```
inline int add(const int x, const int y) {
    return x + y;
}
int main() {
    add(3, 4);
}
```

编译器在编译时，会将函数调用语句 add(3,4)用 add()函数体的代码替换掉（当然 add()函数的局部变量名会做一些调整），这个过程称为**内联展开**。内联展开将用函数的代码替换掉函数调用语句。

如果一个声明为内联函数的函数体中包含了循环语句或者函数体代码比较复杂，编译器在编译时，通常并不会对该内联函数的调用进行内联展开。即，编译器不保证一定会对内联函数调用进行内联展开。

6.7 constexpr

constexpr 关键字可以用来修饰一个变量或函数，表示这个变量或函数的值是编译时可评估的。也就是说变量或函数的值是编译时就能确定的值且不会改变。

constexpr 修饰的变量必须初始化，且初始化表达式必须是一个常量表达式。

constexpr 用于修饰一个函数时，表示这个函数的值可以是一个常量表达式，但也不一定是常量表达式，只有它的参数或返回值表达式是常量表达式时，它才是一个常量表达式。如果 constexpr 函数是一个常量表达式，则可以用它的值对 constexpr 变量进行初始化。

常量表达式就是在编译时能确定值的表达式，可以是一个文字量或者是 const 或 constexpr 修饰的常量表达式或 constexpr 函数。

const 和 constexpr 的区别：

（1）const 的含义是"我保证不变"，用于定义不能被修改的对象，但 const 对象的值通常在程序运行期间才能确定，主要用于如"传给函数的数据不用担心被修改"等接口场合。函数的一个形参声明为 const，表示"我保证不修改"这个形参。如下列函数的形参 a 在函数 f()中是不能被修改的。

```
auto f(const int a) {  /* … */ }
```

（2）constexpr 的含义是"编译时能确定值"，主要用于定义。

- 常量表达式(constant expression)对象，即编译时常量。
- 可返回常量表达式的 constexpr 函数。但不保证一定返回常量表达式。返回常量表达式的 constexpr 函数就是一个常量表达式，否则就是普通的函数。

重温 const：

- 用 const 修饰符定义的变量，称为 **const 对象**。一旦定义后就不能被修改，因此 const 对象必须在定义时就给它一个初始值。

```
const auto PI{3.14};           //const 对象必须在定义时就给它一个初始值
PI = 3.1415926;                //错:不能修改 const 对象
const int ci;                  //错: const 对象必须初始化
```

- const **对象**的初始化式可以是任意复杂的表达式：

```
auto a{2};
const auto ca = a;             //ok: 可以是一个变量
const auto j = 42;             //ok: 可以是一个文字量
const auto k = get_size();     //ok:可以时一个运行时的值
```

- 初始式是常量表达式的 const 对象称为**编译时常量**，否则称为**运行时常量**。

用 const 修饰返回值的函数不是常量表达式，而 constexpr 函数可能是常量表达式也可能不是常量表达式。

读者可以通过下列代码仔细体会 const 和 constexpr 的区别：

```
const auto size() {             //返回一个 const 对象,不是常量表达式
    int i{ 9 };  return i;
}
constexpr auto size1(int x) {   //constexpr 函数可以返回常量表达式
    int i{ 9 };  return i;
}
auto a{3};
const auto b{ 4 };              //编译时常量,因为 4 是常量表达式
const auto c{ size() };         //运行时常量,因为 size()函数值运行时才能确定
const auto d{ size1(a) };       //运行时常量,因 a 是变量,所以 size1(a)不是常量表达式
const auto e{ size1(b) };       //编译时常量,因 b 是常量表达式,所以 size1(b)是常量表达式
char arr[a], arr1[b], arr2[c], arr3[d], arr4[e];  //数组大小必须是常量表达式
```

constexpr 修饰的变量是一个常量表达式（即编译时常量），因此其初始化式也必须是常量表达式。

```
auto a{ 3 };
const auto b{ 4 };              //编译时常量,但不一定是常量表达式
constexpr auto c{ 5 };          //常量表达式,也是编译时常量
constexpr auto d{ c + 1 };      //c 是常量表达式,因此 c + 1 也是,所以 d 也是常量表达式
constexpr auto e{size()};       //错: size()函数返回的不是编译时常量
constexpr auto f{ size1(a) };   //错: size1(a)不是常量表达式,因为 a 不是编译时常量
constexpr auto g{ size1(b) };   //ok: size1(b)是常量表达式,因为 b 是编译时常量
```

另外，constexpr 函数都是 inline（内联）函数。

6.8 实战：二维字符图形库 ChGL

6.8.1 如何在字符终端上绘图

早期的计算机显示终端只能显示字符，即为字符终端，是只能接收和输出文本信息的终端。早期的计算机操作系统也是基于字符的操作系统，如 DOS 操作系统。现代计算机都采用彩色显示器作为显示设备的图形终端，不但可以接收和输出文本信息，也可以输出图形图像，操作系统也是图形用户界面的窗口操作系统，可以绘制各种复杂的图形图像。当然，现代操作系统仍然提供模拟字符终端的终端窗口（也称为控制台窗口），在控制台窗口中只能显示字符。前面的 C++程序的所有输出都是通过代表控制台窗口的 cout 进行的。

那么，如何在控制台窗口绘制曲线或图案呢？

4.5 节的 Pong 游戏用字符来表示游戏场景中的各种元素（背景或运动物体等），为了绘制每一帧游戏画面，采用了一个二层的循环遍历窗口的每个位置，并用条件语句判断每个位置属于哪个对象而输出相应的字符。对于 Pong 这种场景简单的游戏，绘制没什么问题，但如果场景复杂（对象增多、对象不是单一颜色），对每个位置要用大量的条件语句进行判断，代码会越来越复杂，难以管理维护。

彩色显示器是如何显示图形图像的呢？彩色显示器的屏幕实际是由很多像素构成的，即是彩色像素的矩阵（矩形），如图 6-7 所示。彩色显示器屏幕上的像素个数称为分辨率，如 1024 像素×618 像素，只要给每个像素特定的颜色，屏幕就会显示某种图形（图像），同样尺寸的屏幕，分辨率越高显示的图像质量越高。每个显示器还有显示内存，它是用于保存显示图像的内存，即帧缓冲器，屏幕的每个像素在帧缓冲器中都有对应的存储单元（姑且也称为像素），要显示的图形图像先会绘制到帧缓冲器，即给帧缓冲的每个像素设置相应的颜色。这些内容被视频控制器读取，用于控制屏幕上对应像素的颜色。

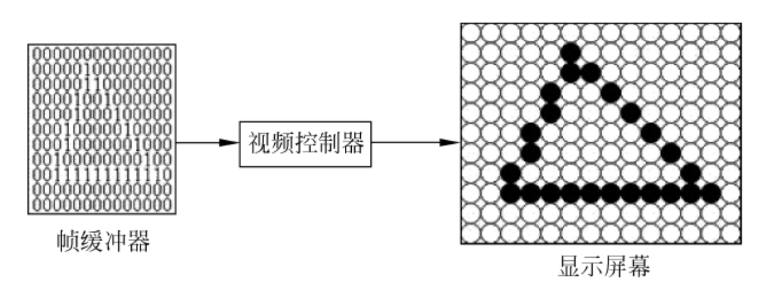


图 6-7 彩色显示器的显示原理

C++程序只有通过和显示驱动程序打交道的图形库，如著名的 OpenGL 图形库，才能在彩色显示器的屏幕上绘制图形，如各种曲线、图像等。

本书将采用模拟在彩色像素显示器上显示图形的过程开发一个模拟图形库的字符图形库。基本思路是：用字符模拟像素，不同字符模拟像素的不同颜色，用数据元素是字符的一块内存模拟帧缓冲器，绘制图形的过程就是给这个字符帧缓冲器的每个位置的所谓字符像素设置不同的字符，然后通过一个模拟视频控制器的显示函数在字符终端上显示出这个字

符像素图像。

OpenGL 等图形库通过提供许多 API 函数给程序员，程序员调用这些 API 函数完成各种复杂的图形绘制任务。因此，为模拟 OpenGL 图形库，本书将设计针对这种字符像素帧缓冲器绘制图形的最简单的字符图形库，即以一些基本函数作为这种字符图形库的 API 函数。在此基础上，可以开发各种字符图形程序，包括前面的游戏程序。

6.8.2 字符图形库 ChGL

本书设计的基于字符的图形库称为 ChGL，采用循序渐进的方法设计这个字符图形库，首先用不同的字符表示不同的颜色(using color = char)，即一种字符就是一种颜色，并用一块 char 类型的指针(color * framebuffer)指向字符帧缓冲器的动态内存。然后是一些 API 函数，如初始化图形窗口(initWindow())，清空窗口(clearWindow())，销毁窗口(destoryWindow())，显示图像(show())，读写缓冲器字符像素颜色(setPixel()、getPixel())，设置/读取清屏颜色 clear_color 的 set_clear_color() / get_clear_color()。

```
using color = char;            //定义一个表示颜色的 color 类型,每种字符就是一种颜色
color * framebuffer{nullptr};  //帧缓冲器
int framebuffer_width,framebuffer_height;
color clear_color{' '};        //清屏颜色

bool initWindow(int width,int height);  //初始化一个窗口,返回 bool 值表示成功还是失败
void clearWindow();            //清空窗口内容
void destoryWindow();          //销毁窗口,释放帧缓冲器占用的内存
void show();                   //显示帧缓冲区的图像
void setPixel(const int x,const int y,color c = ' ');  //设置像素的颜色
color getPixel(const int x,const int y);               //设置像素的颜色
void set_clear_color(color c) { clear_color = c; }
color get_clear_color(){ return clear_color; }
```

下面是上述最基本的函数的实现代码。

```
//初始化一个窗口,返回 bool 值表示成功还是失败
bool initWindow(int width,int height){
    framebuffer = new color[width * height];
    if(!framebuffer) return false;
    framebuffer_width = width;
    framebuffer_height = height;
    clearWindow();
    return true;
}

//用清屏颜色 clear_color 清空窗口内容
void clearWindow(){
    for(int y = 0; y< framebuffer_height;y++)
        for(int x = 0; x< framebuffer_width;x++)
            framebuffer[y * framebuffer_width + x] = clear_color;
```

```
}

//销毁窗口,释放帧缓冲器占用的内存
void destoryWindow(){
     delete[] framebuffer;
     framebuffer = nullptr;
}

//显示帧缓冲区的图像
void show(){
    for(int y = 0; y< framebuffer_height;y++){
        for(int x = 0; x< framebuffer_width;x++)
            std::cout << framebuffer[y * framebuffer_width + x];
        std::cout <<'\n';
    }
}
```

对于一个屏幕窗口，窗口的每个像素通常用(x,y)坐标表示，其中y是从上向下的纵坐标，x是从左到右的横坐标。

frame_buffer 是一个字符 color(char)类型的指针，指向动态分配的内存，用来表示帧缓冲器字符像素矩阵。

假设二维像素矩阵一行一行地存储在这个一维数组中，则位置(x,y)的像素在这个 frame_buffer 表示的一维数组的下标k是多少？

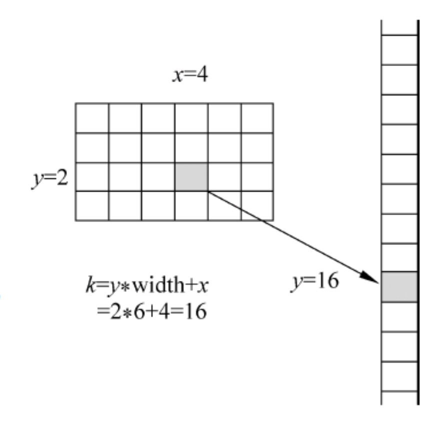


图 6-8 窗口像素坐标到一维存储下标的映射关系

如图 6-8 所示，假设x、y、k下标都是从 0 开始，像素(x,y)前面一共有y行，而每行有 width 个像素，因此前y行一共有$y*$width 个像素，而在像素(x,y)的同一行里，其前面有x个元素，因此，像素(x,y)前面一共有$y*$width$+x$个像素，因此在一维数组中，其对应下标$k=y*$width$+x$。

因此，可以写出下面的根据屏幕窗口坐标读写其对应的一维帧缓冲器对应的像素颜色的代码：

```
void setPixel(const int x,const int y,color c){
    framebuffer[y * framebuffer_width + x] = c;
}
color getPixel(const int x,const int y){
    return framebuffer[y * framebuffer_width + x];
}
```

可以编写一个简单的函数测试一下。

```
int main(){
    if(!initWindow(25,15)){
        return 1;
    }
```

```
    set_clear_color('-');
    clearWindow();
    int x{10},y{10};
    setPixel(x,y,'*');
    setPixel(x-1,y+1,'*');setPixel(x,y+1,' ');setPixel(x+1,y+1,'*');
    setPixel(x-2,y+2,'*');setPixel(x-1,y+2,' ');setPixel(x,y+2,'*');setPixel(x+1,
    y+2,' ');setPixel(x+2,y+2,'*');
    show();
}
```

运行程序,将显示如图 6-9 所示的结果。

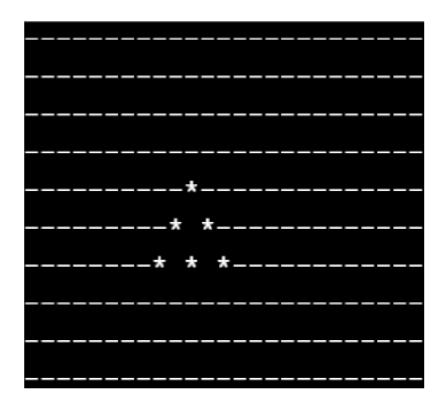


图 6-9 测试 ChGL:输出一个三角形

下面的代码则绘制一个正弦曲线。

```
#include <cmath>
#define SINEHEIGHT  20
#define DEGREESTEP  5
void draw_sin_curve (){
    for (int degree = 0 ; degree < 361 ; degree = degree + DEGREESTEP){
        auto x = floor((degree / DEGREESTEP) + 0.5) +1;
        auto y = floor(  sin(degree * 3.141 / 180)  * (SINEHEIGHT / 2)  + 0.5)
         + SINEHEIGHT / 2 +1;

        setPixel(x,y,'*');
    }
    auto x = 1;
    auto y = SINEHEIGHT / 2 + 1 ;
    setPixel(x,y,'*');
}
int main(){
    if(!initWindow(60,50)){
        return 1;
    }
    draw_sin_curve();
    show();
}
```

运行程序，将显示如图 6-10 所示的结果。

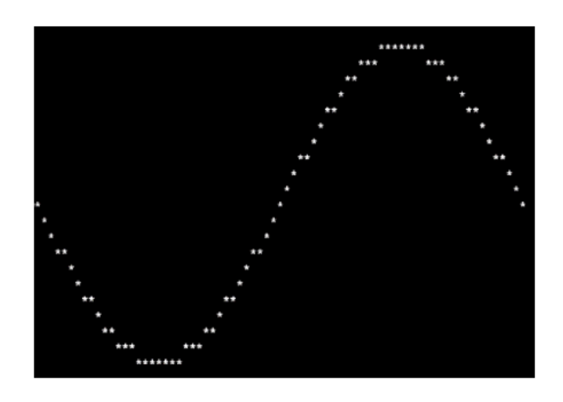


图 6-10 绘制正弦曲线

6.8.3 曲线绘制 API 函数 plot()

许多编程语言如 Matlab、Python 等都提供了非常方便的绘制图像的 API 函数，如 plot() 函数可以用来绘制一条曲线。因此，可以给 ChGL 图形库添加一个这样的 API 函数。为简单起见，本书的 plot()仅仅绘制二维点集，有兴趣的读者可以参考图形学的直线段或曲线段的扫描转换算法编写可以绘制其他曲线的 plot()函数，甚至可以将它扩展成三维图形库。

下面是 plot()函数的代码，辅助函数 min_max()用来求一个数组中的最大、最小值。plot()函数的前 3 个形参 x、y、n 分别是点集的 x、y 坐标数组及数组大小，win_w 和 win_h 是窗口的宽和高，offset 是绘制图形在窗口中的偏移，upset 表示绘制图形是否倒置（因为 setPixel()的 y 在屏幕窗口中是向下的）。

```
inline void min_max(double s[], const int n, double &min, double &max) {
    if (n <= 0) return;
    min = s[0]; max = s[0];
    for (int i = 1; i < n; i++) {
          if (s[i] < min) min = s[i];
          if (s[i] > max) max = s[i];
    }
}

inline void plot(double x[], double y[], const int n, const int win_w, const int win_h,
    const int offset = 2, const bool upset = true) {
    auto plot_w{ win_w - 2 * offset }, plot_h{ win_h - 2 * offset };
    //求 x[],y[]数组中的最大值、最小值
    double x_min, x_max, y_min, y_max, x_dist, y_dist;
    min_max(x, n, x_min, x_max);
    min_max(y, n, y_min, y_max);
    x_dist = x_max - x_min;
    y_dist = y_max - y_min;
    auto scale_x = new double[n], scale_y = new double[n];
    if (!scale_x || !scale_y) return;
    //放缩到绘图 plot 窗口中
    for (int i = 0; i < n; i++) {
```

```
            scale_x[i] = plot_w * (x[i] - x_min) / x_dist;
            scale_y[i] = plot_h * (y[i] - y_min) / y_dist;
    }
    //绘制点集
    if (upset)
            for (int i = 0; i < n; i++)
                    setPixel(scale_x[i] + offset, win_h - (scale_y[i] + offset), '*');

    else
            for (int i = 0; i < n; i++)
                    setPixel(scale_x[i] + offset, scale_y[i] + offset, '*');
    delete[] scale_x;
    delete[] scale_y;
    show();
}
```

plot()函数首先确定绘制图形的矩形区域的宽 plot_w、高 plot_h，然后计算 x 和 y 坐标的最大、最小值，以便将所有坐标点缩放到绘制窗口中，最后用这些缩放坐标(scale_x，scale_y)调用 setPixel()函数绘制这些点，绘制时根据 upset 的值决定是否逆置 y 坐标，并适当偏移。

下面代码将表示房屋面积和房屋价格的几个数据点用 plot()函数绘制出来。

```
int main() {
    const int w = 100, h = 40;
    if (!initWindow(w, h)) {
            return 1;
    }
    double x[] = { 2014, 1600, 2400, 1416, 3000, 3670, 4500 };    //房屋面积
    double y[] = {400, 330, 369, 232, 540, 620, 800};              //房屋价格
    plot(x, y, 7, w, h);
    return 0;
}
```

图 6-11 所示是输出的结果图形。

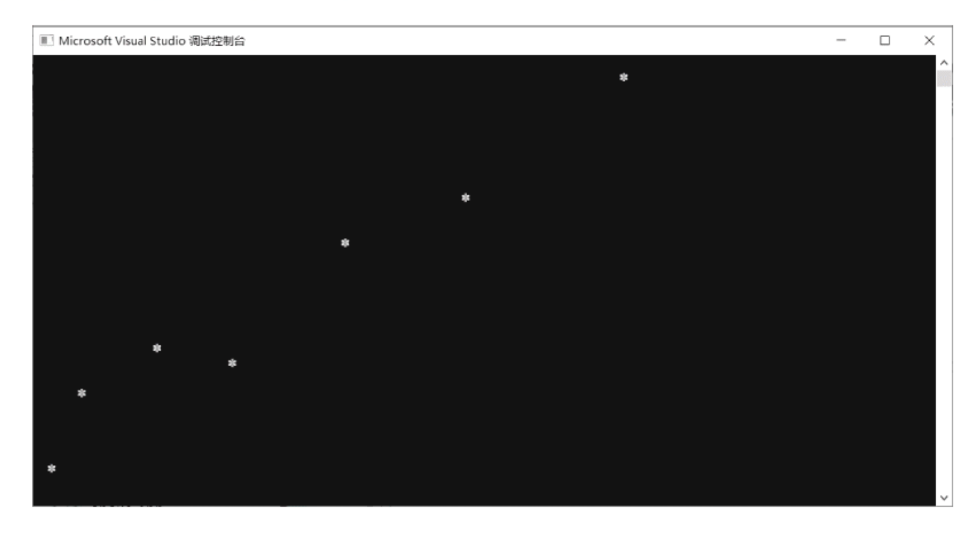


图 6-11　用 plot()函数绘制(房屋面积、房屋价格)的点集

6.9 实战：基于 ChGL 的控制台游戏

在字符图形库 ChGL 的基础上可以绘制各种复杂的图形，当然也可以编写各种控制台游戏。

6.9.1 游戏程序的框架

一个游戏就是一个随时间变化的画面，每一时刻的画面包括背景图像和一些动态物体(称为**精灵**)的图像。游戏一开始会进行一些初始化工作，然后显示开始画面，根据用户的输入和时间流逝，游戏中的元素(对象)会发生变化，从而导致画面产生变化。游戏的过程通常一直循环地"**处理用户输入、更新游戏的数据、绘制场景**"。4.5 节的 Pong 游戏中只有一个 main()主函数，当游戏变得复杂时，这个主函数中的代码会变得臃肿复杂。为此，可以采用分而治之的过程式编程思想，即将一些相对独立的功能用单独的函数表示，如初始化函数、更新游戏状态、事件处理函数、绘制场景函数等，从而可以使得代码结构清晰、易于理解跟踪。

因此，所有游戏具有如下程序结构或框架：

```
int main(){
    //1.初始化
    init();

    //2.游戏循环
    while(running){
        processInput();          //2.1 处理用户输入
        update();                //2.2 更新游戏数据
        renderScene();           //2.3 绘制场景
    }
    return 0;
}
```

其中，用一个 init()函数初始化游戏环境和数据，然后是一个只要 running 为 true 就一直循环的程序块，其中 processInput()、update()、renderScene()分别负责处理用户输入、更新游戏状态、绘制场景。每个函数完成一个专门的工作。

6.9.2 用 ChGL 和函数重写 Pong 游戏

1. 初始化游戏数据

可以定义一些全局变量来表示游戏中的数据，除前面的帧缓冲器相关的数据外，还包括球及左右挡板的位置和速度、双方的得分及其绘制位置。

```
//1. 初始化游戏中的数据
int ball_x, ball_y, ball_vec_x{0}, ball_vec_y{ 0 };     //球的位置和速度
int paddle_w, paddle_h;                                  //挡板的长宽
```

```
int paddle1_x, paddle1_y, paddle1_vec{0};             //左挡板的位置和速度
int paddle2_x, paddle2_y, paddle2_vec{0};             //右挡板的位置和速度
int score1{ 0 }, score2{0}, score1_x, score1_y, score2_x, score2_y;  //得分及得分的显示位置
```

下面的 init()函数对这些数据进行初始化。

```
bool init(const int window_width = 100,const int window_height = 40){
    if (!initWindow(window_width, window_height)) {     //初始化窗口
        return false;
    }
    ball_x = window_width / 2;
    ball_y = window_height / 2;

    paddle_w = 4; paddle_h = 10;
    paddle1_x = 0; paddle1_y = window_height / 2 - paddle_h / 2;
    paddle2_x = window_width - paddle_w; paddle2_y = paddle1_y;
    paddle1_vec = 3;  paddle2_vec = 3 ;

    score1 = 0; score2 = 0;
    score1_x = paddle_w + 8;            score1_y = 2;
    score2_x = window_width - 8 - paddle_w; score2_y = 2;

    srand((unsigned)time(0));                           //生成随机数种子
    random_ball(window_width, window_height);
    return true;
}
```

其中,C 库函数 srand()用于初始化一个随机数发生器,而函数 random_ball()用于初始化球的随机速度,假设球心在中心位置(当然球心也可以是随机位置)。

```
//初始化球的位置和速度
void random_ball(const int window_width, const int window_height) {
    ball_x = window_width / 2; ball_y = window_height / 2;
    ball_vec_x = rand() % 3 + 1;                        //生成一个随机整数表示球的横向速度
    ball_vec_y = rand() % 3 + 1;                        //生成一个随机整数表示球的纵向速度
    if (rand() % 2 == 1) ball_vec_x = - ball_vec_x;     //速度可以是负数
    if (rand() % 2 == 1) ball_vec_y = - ball_vec_y;     //速度可以是负数
}
```

2. 绘制背景

背景包括上下墙壁、左右沟渠和中间分隔线,可以用一个函数将其绘制到画布上:

```
void draw_background() {
    clearWindow();                                      //清空为背景颜色
    int &window_width = framebuffer_width,&window_height = framebuffer_height;
    auto right{ window_width - 1 }, middle{ window_width / 2 };
    for (int y = 0; y < window_height; y++) {
        setPixel(0, y, boundary_color);
        setPixel(middle, y, boundary_color);
```

```
        setPixel(right, y, boundary_color);
    }
    int bottom = window_height - 1;
    for (int x = 0; x < window_width; x++) {
        setPixel(x,0, wall_color);
        setPixel(x,bottom,wall_color);
    }
}
```

3. 绘制精灵(球和挡板)

draw_sprites()绘制场景中的精灵：球、挡板和得分。

```
//用 draw_sprites()绘制场景中的精灵: 球、挡板和得分.
void draw_sprites() {
    //绘制球
    setPixel(ball_x, ball_y, ball_color);
    //绘制左、右挡板
    for (auto y = paddle1_y; y < paddle1_y + paddle_h; y++)
        for (auto x = paddle1_x; x < paddle1_x + paddle_w; x++)
            setPixel(x,y, paddle_color);

    for (auto y = paddle2_y; y < paddle2_y + paddle_h; y++)
        for (auto x = paddle2_x; x < paddle2_x + paddle_w; x++)
            setPixel(x,y, paddle_color);

    //绘制分数: 分数是一个字符串
    std::string s1{ std::to_string(score1) }, s2{ std::to_string(score2) };
    for (auto i = 0; i < s1.size(); i++)
        setPixel(score1_x + i, score1_y, s1[i]);
    for (auto i = 0; i < s2.size(); i++)
        setPixel(score2_x + i, score2_y, s2[i]);
}
```

4. 绘制场景

render_scene()用于在画布上绘制场景(背景和精灵)并在屏幕上显示场景。在每次绘制场景前必须用 gotoxy(0,0)来清屏，并且调用 hideCursor()函数隐藏光标。

```
void gotoxy(int x, int y) {
    COORD coord = {x, y };
    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), coord);
}
void hideCursor() {
    CONSOLE_CURSOR_INFO cursor_info = { 1, 0 };
    SetConsoleCursorInfo(GetStdHandle(STD_OUTPUT_HANDLE), &cursor_info);
}
void render_scene() {
    gotoxy(0, 0);              //定位到(0,0),相当于清空屏幕
    hideCursor();
```

```
    draw_background();          //在画布上绘制背景
    draw_sprites();             //在画布上绘制精灵
    show();                     //在屏幕上显示画布内容(场景)
}
```

5. 头文件

当然在程序开头要包含相应的头文件。

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <windows.h>
#include <conio.h>
#include <string>
```

6. 测试

运行下面的程序,将显示一个背景窗口。

```
int main() {
    if (!init()) {
        std::cout << "初始化窗口失败!\n";
        return 1;
    }
    render_scene();
    return 0;
}
```

7. 事件处理

可将事件处理功能封装在一个函数里:

```
int processInput() {
    //处理事件
    char key;
    if (_kbhit()) {
          key = _getch();
          if (key == 27) return -1;
          else if ((key == 'w' || key == 'W') && paddle1_y > paddle1_vec)
                paddle1_y -= paddle1_vec;
          else if ((key == 's' || key == 'S') && paddle1_y + paddle1_vec + paddle_h < HEIGHT)
                paddle1_y += paddle1_vec;
          else if (key == 72 && paddle2_y > paddle2_vec)
                paddle2_y -= paddle2_vec;
          else if ((key == 80) && paddle2_y + paddle2_vec + paddle_h < HEIGHT)
                paddle2_y += paddle2_vec;
    }
    return 0;
}
```

8. 更新游戏状态(数据)

更新球的位置,检测球与墙壁、挡板是否发生碰撞。

```
void update() {
    //2. 更新数据
    ball_x += ball_vec_x;
    ball_y += ball_vec_y;
    if (ball_y < 0|| ball_y >= HEIGHT) {
          ball_vec_y = - ball_vec_y;
          ball_y += ball_vec_y;
    }
    if (ball_x < paddle_w&& ball_y >= paddle1_y && ball_y < paddle1_y + paddle_h) {
          ball_vec_x = - ball_vec_x;
          score1 += 1;
    }
    else if (ball_x > WIDTH - paddle_w && ball_y >= paddle2_y && ball_y < paddle2_y + paddle_h)
    {
          ball_vec_x = - ball_vec_x;
          score2 += 1;
    }
    bool is_out{ false };
    if (ball_x < 0) { score2 += 1; is_out = true; }
    else if (ball_x >= WIDTH) { score1 += 1; is_out = true; }
    if (is_out) {
          random_ball();
    }
}
```

9. main()函数

```
int main() {
    //1. 初始化数据
    init();
    //2.  游戏循环
    while (true) {
        if (processInput() < 0)break;
        update();
        render_scene();
    }
    return 0;
}
```

完整程序请在作者的网站 https://a.hwdong.com 上下载。

6.10 实战：机器学习-线性回归

6.10.1 机器学习

机器学习(maching learning)就是用某种学习算法从经验数据中发现规律，再将这个规律用于新的情况的判断、决策和预测。如气象预报部门根据以往的经验(大量数据)建立气象预报模型，再用这个模型对新的天气情况进行预报(预测)。下棋算法根据大量棋盘对局的胜负情况构建下棋算法模型，用于指导下棋。医疗诊断系统可以根据大量病人的各种检查指标和其是否患有癌症的信息建立一个肿瘤诊断模型，这个模型可以用来根据新人的检查指标作出肿瘤诊断。如果有许多不同年龄人的照片，机器学习也可以从这些照片和年龄的对应关系中学习一个模型来对一张新人照片预测其年龄。同样，如果有一组房屋属性(如房屋面积)和房屋价格的数据，也能学习一个房屋属性和价格的关系模型，用来对一个新的房屋预测它的价格。

人工智能主要分为：

- 基于规则的逻辑推理，其中的一个典型代表就是专家系统。它主要根据专家的经验提炼出一些语义规则，然后用这些规则进行逻辑推理，因为需要用到专家的经验知识，所以也可以称为第一代机器学习。
- 基于统计模型的机器学习。它采用一些统计模型如支持向量机(support vector machine，SVM)、核方法(kernel methods)、随机森林、线性或逻辑回归模型、神经网络模型等表示数据中潜在规律的模型，并根据大量数据样本学习出某种假设模型(神经网络或 SVM 模型)的参数。再用这个模型对新的数据进行预测。基于统计模型的机器学习就是人们常说的机器学习，也可以称为第二代机器学习。

深度学习是基于深度神经网络的机器学习，也是目前取得极大成功的人工智能的核心技术。本节介绍的机器学习中的**线性和逻辑回归**是(深度)神经网络的原子和基础，其求解算法如梯度下降法也是(深度)神经网络的核心求解算法。

根据用于学习的数据集中的数据样本是否具有明确的答案，机器学习通常又分为**监督学习**和**非监督学习**。监督学习中每个样本除数据特征外都有明确的答案，如一张人脸照片，其数据特征就是图像上所有像素点的颜色信息，而这张照片表示的人的年龄就是答案(或目标)。非监督学习中，数据样本只有数据的特征，没有明确的答案。如有一个照片集合，但每张照片没有明确的目标，希望找出这组照片具有的某些规律，如可以用聚类算法将它们分为男人和女人 2 组，也有可能按照肤色将它们分成不同人种的照片。这种在没有明确答案的数据集中寻找某种规律的学习称为**非监督学习**。

线性回归(如从照片预测年龄或从房屋面积预测其价格)和逻辑回归(从医学指标判断是否是肿瘤)都属于监督学习。**线性回归**用于预测一个连续值，即预测的结果是一个连续值(如房屋价格)；**逻辑回归**用于分类，即确定一个数据是几种类别中的哪一类(如是肿瘤还是非肿瘤)。

6.10.2 假设函数、回归和分类

6.8.3 节最后例子给了一个一组房屋面积及其价格并用 plot()函数绘制了其图形(见图 6-11)。

根据这组价格已知的房屋数据,对于一个面积已知的新房屋,能否预测其价格？这是一个监督学习问题：先用这组数据学习某个模型,再用这个模型预测新房屋的价格。从数学角度来描述,监督学习实际上就是学习一个数学函数 $y = h(x)$,其中的数据面积称为自变量或特征,用 x 表示,而价格称为因变量或目标变量,用 y 表示,$h(x)$是表示 x 和 y 关系的某种数学函数(如线性函数、二次函数或更加复杂的函数,如深度神经网络表示的函数)。这个函数也称为**“假设函数”**。

数据集中的每个数据(x^i,y^i)称为一个样本。如果要预测的目标变量 y 是一个连续的值,这种监督学习称为回归；如果要预测的目标变量 y 是一个离散的值,这种监督学习称为**分类**。

假设函数 $h(x)$的集合通常是一个无穷集合,但可以用一组未知参数刻画这些函数,如 $y=h(x) = ax+b$,不同的 a、b 参数就表示一个不同的函数,回归的目标就是如何根据一组数据在某种最佳的意义上求出一个参数(如 a、b)确定的假设函数,即确定这些未知参数。

6.10.3 线性回归

1. 线性回归的定义

如果表示目标变量 y 和特征 x 之间的假设函数 $h(x)$是一个线性函数,这种监督学习称为**线性回归**(linear regression)。即线性函数 $h(x)$表示的是一个直线。对于一个样本,将其特征 x 代入这个假设函数 $h(x)$就得到样本 x 的目标值(预测值)：

$$h_\theta(x) = \theta_0 + \theta_1 * x$$

x 和 y 之间的这个线性假设函数对应到二维平面上的图像就是一个直线,不同的参数 θ_0 和 θ_1 对应不同的直线,线性回归就是要求解一个最佳的假设函数(直线),使得所有训练数据集中的样本$\{x^i,y^i\}$和这个最佳的直线最接近,当然样本点不会正好都位于这个最佳直线上。

何为最佳直线?一种简单的办法是对每个样本$\{x^i,y^i\}$,用假设函数预测得到的 $h_\theta(x^i)$和目标值 y^i 的误差$(y^i-h_\theta(x^i))^2$ 作为这个样本的误差,线性回归学习的目标是使所有样本的误差之和 $\sum_i(y^i-h_\theta(x^i))^2$ 为最小。这就是人们熟悉的**“最小二乘问题”**,即求使最小二乘代价最小的参数 $\theta=(\theta_0,\theta_1)$。对于这个代价函数,乘以任意常数都不会改变这个最小二乘问题的解,一般采用的是如下的代价函数：

$$J(\theta_0,\theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(y^i-h_\theta(x^i))^2$$

其中,m 是样本的数目。线性回归就是求解最佳的 $\theta=(\theta_0,\theta_1)$使得上述代价函数值最小：

$$\min_{\theta_0,\theta_1} J(\theta_0,\theta_1)$$

这个代价函数 $J(\theta_0,\theta_1)$是多个未知参数(θ_0,θ_1)的函数,即是一个多变量的函数,这个多变量函数的最小值问题(求解 θ)通常有 2 种解法：

(1) 正规方程(normal equation)。

(2) 梯度下降法(gradient descent)。

2. 假设函数 $h_\theta(x)$的向量表示

上述例子中样本的特征 x 只有一个特征值,实际应用中,一个样本的特征通常包含多个不同的特征值,如房屋预测问题,一个房屋的特征值可能包含房屋面积、房间个数等多个特征信息。疾病诊断中,通常需要检查多个指标,即一个样本的特征是多个特征值。可将这些特征值表示为一个向量形式,如一个样本的 2 个特征值 x_1、x_2,可表示为 $\boldsymbol{x}=(x_1,x_2)$,线性假设函数 $h_\theta(x)$可写成:

$$h_\theta(x)=\theta_0+\theta_1x_1+\theta_2x_2$$

为了将 $h(x)$写成更简单的统一的向量形式,通常人为地给样本添加一个特征值 1,即将 $\boldsymbol{x}=(x_1,x_2)$写成 $\boldsymbol{x}=(1,x_1,x_2)$。另外,按照数学向量的习惯写法,一般在进行数学推导时都写成列向量而不是行向量形式,即:$\boldsymbol{x}=(1,x_1,x_2)^\mathrm{T}$,$\boldsymbol{\theta}=(\theta_0,\theta_1,\theta_2)^\mathrm{T}$。它们的转置就是行向量:$\boldsymbol{x}^\mathrm{T}=(1,x_1,x_2)$、$\boldsymbol{\theta}^\mathrm{T}=(\theta_0,\theta_1,\theta_2)$。自然地,假设函数 $h_\theta(x)$可写成向量形式:

$$h_\theta(x)=\theta_0+\theta_1x_1+\theta_2x_2=\boldsymbol{x}^\mathrm{T}\boldsymbol{\theta}=\boldsymbol{\theta}^\mathrm{T}\boldsymbol{x}$$

6.10.4 多变量函数的最小值、正规方程

有一个函数 $J(\theta_0,\theta_1,\cdots,\theta_n)$,目标是 $\min\limits_{\theta_0,\theta_1,\cdots,\theta_n} J(\theta_0,\theta_1,\cdots,\theta_n)$。

根据数学分析(高等数学)知识,函数的最值点$(\theta_0,\theta_1,\cdots,\theta_n)$的必要条件是函数在该点的导数为 0(对于多变量,即其梯度为 0)。

代价函数 $J(\theta)$是许多样本的误差累加和,对于其中的一个样本,其关于参数 θ_j 的导数是:

$$\begin{aligned}\frac{\partial(h_\theta(x^i)-y^i)^2}{\partial\theta_j}&=2(h_\theta(x^i)-y^i)\frac{\partial((\theta_0*x_0^i+\theta_j*x_j^i+\cdots+\theta_n*x_n^i)-y^i)}{\partial\theta_j}\\&=2(h_\theta(x^i)-y^i)*x_j^i\end{aligned}$$

根据导数的加法性质,整个代价函数关于参数 θ_j 的导数就是每个样本关于参数 θ_j 的导数之和的平均:

$$\frac{\partial J(\theta_0,\theta_1,\cdots,\theta_n)}{\partial\theta_j}=\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^i)-y^i)*x_j^i(1)$$

对于每个 θ_j,令导数等于 0,就得到一个关于 $\theta_0,\theta_1,\cdots,\theta_j,\cdots,\theta_n$ 的 n 个方程组成的方程组。

$$\begin{cases}\sum\limits_{i=1}^{m}(h_\theta(x^i)-y^i)*x_0^i=0\\\vdots\\\sum\limits_{i=1}^{m}(h_\theta(x^i)-y^i)*x_j^i=0\\\vdots\\\sum\limits_{i=1}^{m}(h_\theta(x^i)-y^i)*x_n^i=0\end{cases}\qquad(2)$$

可以用矩阵和向量简洁表示上述方程组。令

$$
\boldsymbol{X}=\begin{pmatrix} x_0^1 & x_1^1 & \cdots & x_j^1 & \cdots & x_n^1 \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ x_0^i & x_1^i & \cdots & x_j^i & \cdots & x_n^i \\ \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\ x_0^m & x_1^m & \cdots & x_j^m & \cdots & x_n^m \end{pmatrix}=\begin{pmatrix} x^{1\mathrm{T}} \\ \vdots \\ x^{i\mathrm{T}} \\ \vdots \\ x^{m\mathrm{T}} \end{pmatrix} \qquad \boldsymbol{Y}=\begin{pmatrix} y^1 \\ \vdots \\ y^i \\ \vdots \\ y^m \end{pmatrix} \qquad \boldsymbol{\theta}=\begin{pmatrix} \theta^0 \\ \vdots \\ \theta^j \\ \vdots \\ \theta^n \end{pmatrix}
$$

上述方程组，可以写成矩阵和向量的形式：

$$\boldsymbol{X}^{\mathrm{T}}(\boldsymbol{X}\boldsymbol{\theta}-\boldsymbol{Y})=0$$

即

$$\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{\theta}=\boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y}$$

这就是所谓的**正规方程**(normal equation)。$\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$ 是一个 $n\times n$ 的矩阵，两边乘上其逆矩阵，从而得到方程组的解：

$$\boldsymbol{\theta}=(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y}$$

虽然可以用正规方程方法直接求出线性回归的解$\boldsymbol{\theta}$，如果样本个数即 m 很大或者$\boldsymbol{\theta}$ 向量的参数多，这种矩阵乘积或求矩阵的逆矩阵计算量很大，效率比较低。因此，一般就采用**迭代法**求方程组的解，其中最常用的方法就是"**梯度下降法(gradient descent)**"：从一个$\boldsymbol{\theta}$ 的初始值出发，沿着梯度方法迭代更新未知参数$\boldsymbol{\theta}$。

6.10.5 梯度下降法

梯度下降算法(gradient descent algorithm)就是从一个初始的$\boldsymbol{\theta}$ 值，迭代的沿着 J 关于$\boldsymbol{\theta}$ 的梯度的反方向前进(即更新$\boldsymbol{\theta}$ 值)，不断逼近最佳的$\boldsymbol{\theta}$。

- 随机选择一组值作为$\boldsymbol{\theta}$ 的初始值。
- 循环迭代直至结果收敛：

$$\theta_j=\theta_j-\alpha\frac{\partial J(\theta_0,\theta_1,\cdots,\theta_n)}{\partial\theta_j}$$

其中，α 是学习率，表示更新$\boldsymbol{\theta}$ 的速度，数值太小收敛缓慢，数值太大可能会跳过最佳$\boldsymbol{\theta}$，导致$\boldsymbol{\theta}$ 值来回振荡。一般来说，这个学习率不是固定的，先开始可以取较大的值，加快更新速度，然后逐渐减小，以提高收敛性。

如图 6-12 所示，随着$\boldsymbol{\theta}$ 的更新，其对应的函数值 $J(\boldsymbol{\theta})$ 也在减小(下降)。

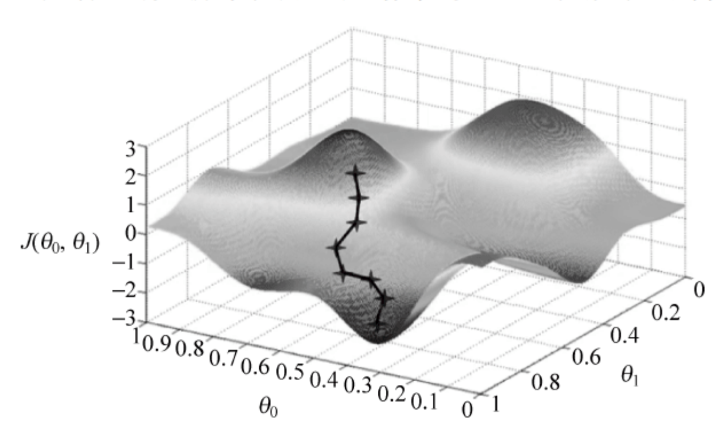


图 6-12 沿梯度反方向更新$\boldsymbol{\theta}$，使函数值不断下降(来自网络图片)

因为

$$\frac{\partial(h_\theta(x^i)-y^i)^2}{\partial\theta_j}=2(h_\theta(x^i)-y^i)\frac{\partial((\theta_0*x_0^i+\theta_j*x_j^i+\cdots+\theta_n*x_n^i)-y^i)}{\partial\theta_j}$$
$$=2(h_{\boldsymbol{\theta}}(x^i)-y^i)*x_j^i$$

因此

$$\frac{\partial J(\theta_0,\theta_1,\cdots,\theta_n)}{\partial\theta_j}=\frac{1}{m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(x^i)-y^i)*x_j^i$$

因此,迭代公式为

$$\theta_j=\theta_j-\alpha\frac{1}{m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(x^i)-y^i)*x_j^i$$

图 6-13 所示是参数$\boldsymbol{\theta}$的一个典型的更新过程,其中每个圆圈上的$\boldsymbol{\theta}$对应的$J(\boldsymbol{\theta})$都是等值的。

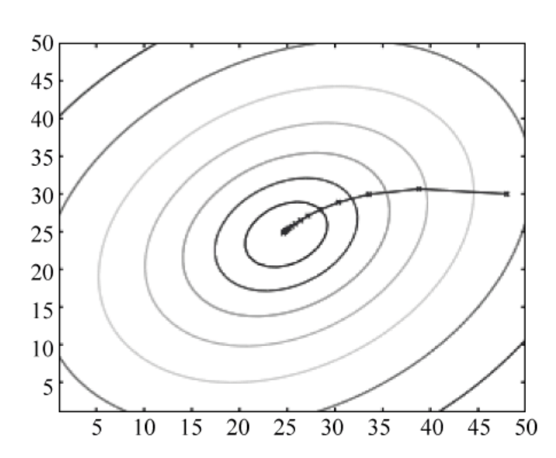


图 6-13 $h_{\boldsymbol{\theta}}(x)=\theta_0+\theta_1*x_1+\theta_2*x_2$ 的迭代过程示意图(来自网络图片)

6.10.6 梯度下降法求解线性回归问题:模拟数据

网上很多线性回归的文章和代码基于第三方库,这些库掩盖了算法的实现细节,本书的线性回归不借助任何第三方库,只利用 C++ 自身的语言特性,有助于读者更好地理解算法原理和实现细节。

首先,模拟生成一组样本数据,该样本数据是对线性函数如 $y=\theta_0+\theta_1*x$ 的随机噪声取样(如图 6-14 所示):

```
double X[][1]{
    9.16481174938805,  3.9617176989763605,  1.9118988617843014,
    4.770872353143195,  8.96268633959237,  4.347497877496233,
    0.7837488406009996, 9.003451281535993,  9.219537986787007,
    0.14895852444561486 };
double Y[]{ 19.3296234987761, 8.923435397952721, 4.823797723568603,
    10.54174470628639,18.92537267918474, 9.694995754992465, 2.567497681201999,
    19.00690256307198б,19.439075973574013, 1.2979170488912297};
```

在下面的线性回归的实现程序中,编写了如下的一些函数。

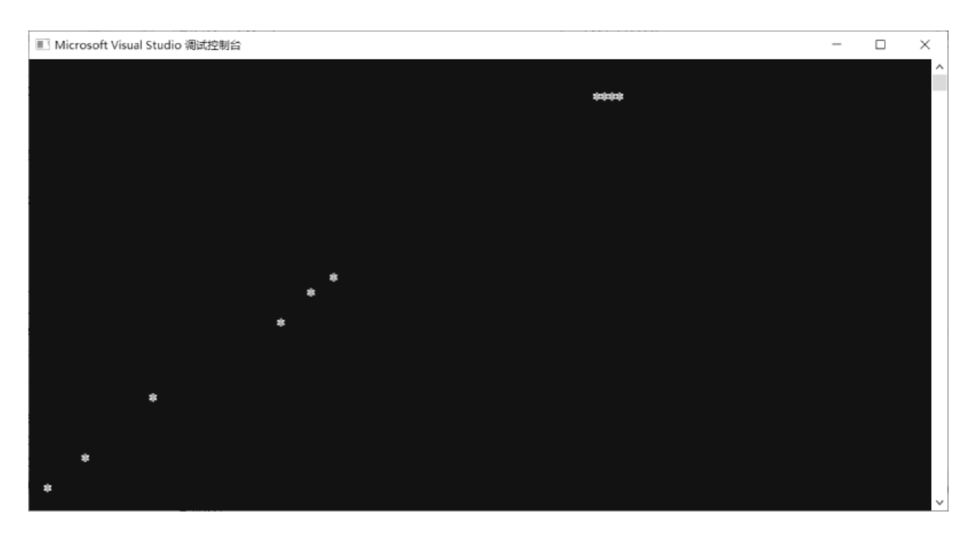


图 6-14 二维平面上的采样点

- h_theta_x()：对一个 x，计算假设函数 $h_\theta(x)$的值。
- cost_function()：计算代价函数 $J(\theta)$及其梯度的值。
- gradient_descent()：梯度下降算法更新 θ。
- main()：传入数据，调用 gradient_descent()求解 θ。

h_theta_x()对于 m 个样本的每个样本 x 计算其假设函数的值 $h_\theta(x)$。cost_function()的返回值是代价函数的值，而 g[i]则是代价函数相对于每个参数 θ_i 的导数。gradient_descent()根据计算的梯度值 g 更新 θ(theta[j] -= alpha * g[j])。

```
const int m{ 10 }, n{2};                    //m 是样本数目,n 是 theta 参数个数
auto h_theta_x(const double h[],const double X[][n], const double theta[],const int m){
    auto cost{0.};
    for (int i = 0; i < m; i++)
        h[i] = h_theta_x(X[i], theta, n);
}
auto cost_function(double g[], const double X[][n], const double * Y, const double theta[],
const int m){
    auto f{ 0. };
    for (auto j = 0; j != n; j++) g[j] = 0.;
    for (int i = 0; i < m; i++) {
        auto h = h_theta_x(X[i], theta, n);
        auto h_y{ h - Y[i] };
        f += (h_y * h_y);                   //累加误差
        //累加梯度
        for (int j = 0; j < n; j++)
            g[j] += (h_y * X[i][j]);
    }
    f /= (2 * m);                           //平均误差
    for (int j = 0; j < n; j++)             //平均梯度
        g[j] /= m;
    return f;
}
```

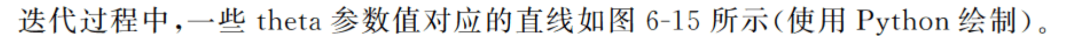
```
auto gradient_descent(double X[][n], double * Y, double theta[],
    double alpha, const int iterations, const int m,double * cost_history){
    for (auto iter = 0; iter != iterations; iter++) {
        double g[n]{};
        auto f = cost_function (g, X, Y, theta, m);
        cost_history[iter] = f;
        for (auto j = 0; j < n; j++)      //更新 theta
            theta[j] -= alpha * g[j];
    }
}
int main() {
    //准备训练数据,为数据特征增加一列 1
    double train_X[m][n]{}, train_Y[m]{};
    for (int i = 0; i < m; i++) {
        train_X[i][0] = 1;
        train_X[i][1] = X[i][0];
        train_Y[i] = Y[i];
    }
    double theta[n]{ };                         //未知参数
    auto alpha(0.0001);
    auto iterations{ 1000 };
    double cost_history[1000];
    gradient_descent(train_X, train_Y, theta, alpha, iterations, m, cost_history);

    for (int j = 0; j != n; j++)
        cout << theta[j] << '\t';
    cout << '\n';

    for (int j = 0; j!= 1000; j++)
        if(j % 99 == 1)
        cout << j <<'\t'<< cost_history[j] << '\n';
    cout << '\n';
    return 0;
}
```

运行程序,输出最终的 θ(theta)值和迭代过程的部分代价函数值。

```
0.303165    2.06607
1           93.7888
100         40.7122
199         17.6882
298         7.70047
397         3.36781
496         1.48825
595         0.672813
694         0.318981
793         0.165389
892         0.0986588
991         0.0696087
```



迭代过程中，一些 theta 参数值对应的直线如图 6-15 所示（使用 Python 绘制）。

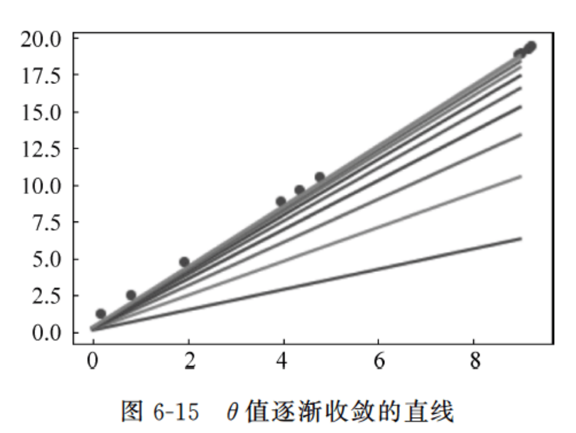


图 6-15　θ 值逐渐收敛的直线

图 6-16 是迭代每隔 99 次的代价函数值的变化情况，可以看出算法逐渐收敛的过程。

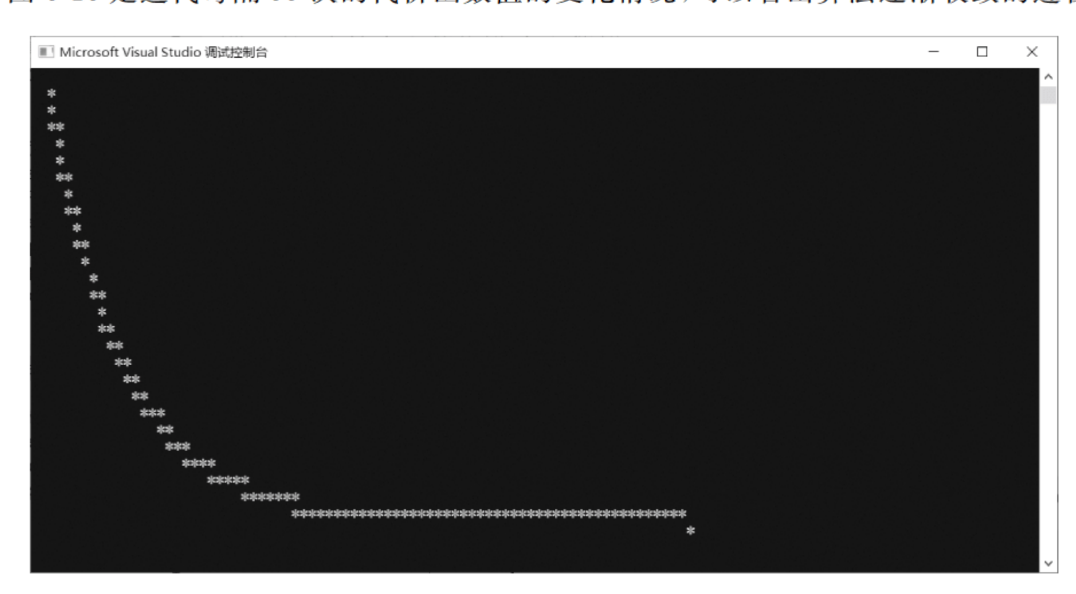


图 6-16　代价函数值不断下降（算法逐渐收敛）

可以看到当学习率固定为 0.0001 时，经过大约 600 次迭代就基本收敛了，可以调整这个学习率，选择一个最佳的学习率保证收敛速度和解的准确性。更好的方法是迭代过程中不断调低学习率。

如果发现梯度下降法不收敛，除调整学习率外，更应该检查梯度的计算是否正确。为此，可以根据导数的定义，即导数是函数的变化率，用如下公式估计梯度，然后和分析梯度两者进行比较，当 ε 很小时，两者应该很接近。

$$\frac{\partial J}{\partial \theta}=\lim_{\varepsilon \to 0}\frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon}$$

为此，可以编写如下代码来检查分析梯度的计算是否正确。

```
//计算代价函数值
auto J(double X[][n], double *Y, double theta[], const int m) {
    double f{ 0. };
    for (int i = 0; i < m; i++) {
```

```
            auto h = h_theta_x(X[i], theta, n);
            auto h_y{ h - Y[i] };
            f += h_y * h_y;            //累加误差
        }
        f /= (2 * m);                  //平均误差
        return f;
}
//数值梯度
auto computeNumericalGradient(double * grad_approx, double X[][n], double * Y, const double
theta[],const int m, double epsilon = 1e-7){
    for (auto j = 0; j < n; j++) {
        double theta_plus[n]{};
        double theta_minus[n]{};
        for (int k = 0; k != n; k++) {
            theta_plus[k] = theta[k];
            theta_minus[k] = theta[k];
        }
        theta_plus[j] = theta[j] + epsilon;
        auto J_plus = J(X, Y, theta_plus, m);
        theta_minus[j] = theta[j] - epsilon;
        auto J_minus = J(X, Y, theta_minus, m);
        grad_approx[j] = (J_plus - J_minus) / (2 * epsilon);
    }
}
```

然后修改前面的 main()函数来检查分析和数值梯度是否一致。

```
int main() {
    //准备训练数据,为数据特征增加一列 1
    double train_X[m][n]{}, train_Y[m]{};
    for (int i = 0; i < m; i++) {
        train_X[i][0] = 1;
        train_X[i][1] = X[i][0];
        train_Y[i] = Y[i];
    }
    double theta[n]{0.5,1.1};          //未知参数

    double g[n]{}, g_[n]{};            //g 是分析梯度,g_是数值梯度
    cost_function(g, train_X, train_Y, theta, m);
    computeNumericalGradient(g_, train_X, train_Y, theta, m);

    for (int j = 0; j != n; j++)
        cout << g[j] << '\t' << g_[j] << '\n';
    return 0;
}
```

程序运行结果如下：

```
-5.67896        -5.67896
-40.0356        -40.0356
```

可以看出，分析梯度和数值梯度几乎是一样的。注意：只有 theta 参数对应直线逼近最佳直线时，两个梯度才几乎相等。

6.10.7 批梯度下降法

如果数据集很大，每一次梯度更新需要用所有数据计算代价和梯度，一方面比较耗时，另一方面如果数据集很大，内存可能无法全部存放，因此，一般采用**批梯度下降法**（**batch gradient descent**）进行梯度更新。也就是每一次只用部分数据来计算代价和梯度并更新参数。假设数据集样本总数是 M，每次从 M 个样本中随机选取 m（$m << M$）个样本做梯度更新。

假如上述的 gradient_descent()是针对 m 个样本的梯度更新，可以在其外层再定义一个函数 batch_gradient_descent ()用来随机选取 m 个样本，然后交给 gradient_descent()用这 m 个样本进行梯度更新。

```
#include <cstdlib>
#include <ctime>
//随机从 M 个样本中选出 m 个
auto batch_data(double train_X[][n], double train_Y[], double X[][n], double Y[],
const int M, const int m) {
    for (auto i = 0; i != m; i++) {
        int s = rand() % M;
        for (unsigned int j = 0; j != n; j++) {
            train_X[i][j] = X[s][j];
        }
        train_Y[i] = Y[s];
    }
}

auto batch_gradient_descent(double train_X[][n], double train_Y[],
double X[][n], double *Y, double theta[],
double alpha, const int batch_iterations, const int iterations,
const int M, const int m, double *cost_history){
    for (auto iter = 0; iter != batch_iterations; iter++) {
        batch_data(train_X, train_Y, X, Y, M, m);
        double * cost_history_ = cost_history + iter * iterations;
        gradient_descent(train_X, train_Y, theta, alpha, iterations, m, cost_history_);
    }
}

int main() {
    //准备训练数据,为数据特征增加一列 1
    double train_X[m][n]{}, train_Y[m]{}, X0[M][n]{};
    for (int i = 0; i < m; i++) {
        X0[i][0] = 1;
        X0[i][1] = X[i][0];
    }
    double theta[n]{ };                    //未知参数
    auto alpha(0.0001);
```

```
    auto batch_iterations(4);
    auto iterations{ 500 };
    double cost_history[500 * 4];
    srand(static_cast<unsigned int>(time(0)));
    batch_gradient_descent(train_X, train_Y,X0,Y, theta, alpha,
                           batch_iterations, iterations,M,m, cost_history);
    for (int j = 0; j != n; j++)
        cout << theta[j] << '\t';
    cout << '\n';

    for (int j = 0; j != 500 * 4; j++)
        if (j % 99 == 1)
            cout << j << '\t'<< cost_history[j] << '\n';
    cout << '\n';
    return 0;
}
```

运行程序，输出结果：

```
0.270615        2.05911
1               118.508
100             76.9794
199             56.8652
298             47.123
397             42.4043
496             40.1188
595             11.0546
694             11.0546
793             11.0546
892             11.0546
991             11.0546
1090            39.2079
1189            39.1203
1288            39.0457
1387            38.9821
1486            38.9279
1585            74.3275
1684            74.1142
1783            74.0137
1882            73.9665
1981            73.9442
```

可以看到也得到差不多的收敛结果。对于数据量很大，用所有数据的梯度下降会很慢，采用批梯度下降法可以大大提高收敛速度并减小计算量。

6.10.8 房屋价格预测

下面使用的是斯坦福大学的公开课程——机器学习课程的房屋预测问题中的数据，这个数据中有 47 个样本，每个样本包含面积、房间数、价格，即样本的特征是面积、房间数，而

目标是价格。数据通常存储在文件中，因为未介绍文件读写，这里直接将数据放在一个二维数组变量中：

```
double house_data[][3]{
2104,3,399900,
1600,3,329900,
2400,3,369000,
1416,2,232000,
3000,4,539900,
1985,4,299900,
…}
```

该数组的完整数据请在作者网站(a.hwdong.com)上下载。

为使用前面的线性回归代码，将这个数据(XY)分成特征和目标 2 个数组(X 和 Y)：

```
auto get_data(double X[][n], double * Y, double XY[][n], const int m) {
    for (auto i = 0; i != m; i++) {
        X[i][0] = 1;
        for (auto j = 0; j != n - 1; j++) {
            X[i][j + 1] = XY[i][j];
        }
        Y[i] = XY[i][n - 1];
    }
}
```

6.10.9 样本特征的规范化

由于一个样本的不同特征(面积、房间数)在数值上具有不同的尺度，如面积的值很大，而房间数很小，如果直接用这些特征值进行机器学习，学习算法会严重倾向于尺度大的特征(即这里的面积)，为了使不同特征具有同等的作用，需要对这些特征进行规范化，即将它们变换到同样的数值范围内(如[0,1]或[−1,1])。对一个特征的规范化过程很简单：首先需要计算所有样本关于这个特征的平均值，再计算所有样本的这个特征围绕平均值的偏移程度(即标准差)，最后将所有样本的这个特征减去其平均值并除以标准差。

$$x \leftarrow= \frac{x - \text{mean}(x)}{\text{stddev}(x)}$$

其中的 mean(x)计算 x 数组中所有特征的平均值，而 stddev(x)是特征的标准差，上述公式可以将所有特征值变换到[−1,1]。

如有一组特征值{−5, 6, 9, 2, 4}，其平均值 mean 为：

$$\text{mean} = (-5+6+9+2+4) / 5 = 3.2$$

将所有特征值减去这个平均值，得到偏差，并计算这些偏差的平方：

$$(-5-3.2)^2 = 67.24$$

$$(6-3.2)^2 = 7.84$$

$$(9-3.2)^2 = 33.64$$

$$(2-3.2)^2 = 1.44$$

$$(4-3.2)^2 = 0.64$$

然后,可以计算出标准差 Stddev:

Stddev = sqrt ((67.24 + 7.84 + 33.64 + 1.44 + 0.64) / 5) = 4.71

最后,所有特征的偏差除以这个标准差,得到规范化后的特征值:

x => (x − mean) / Stddev

结果为:

−5 => (−5 − 3.2) / 4.71 = −1.74

6 => (6 − 3.2) / 4.71 = 0.59

9 => (9 − 3.2) / 4.71 = 1.23

2 => (2 − 3.2) / 4.71 = −0.25

4 => (4 − 3.2) / 4.71 = 0.17

下面是3个辅助函数,对样本的某个特征(某一列)进行规范化:

```
//计算特征的平均值,col 表示 X 的某个特征对应的某一列
auto mean(double X[][n], const int m, const unsigned int col) {
    double mean_value{ 0 };
    for (auto i = 0; i != m; i++)
        mean_value += X[i][col];
    return mean_value /= m;
}
//计算特征的标准差,col 表示 X 的某个特征对应的某一列
auto standard_deviation(double X[][n],  const int m,
    const unsigned int col,  const double mean_value){
      double sd{ 0 };
      for (auto i = 0; i != m; i++) {
           auto diff{ (X[i][col] - mean_value) };
           sd += diff * diff;
      }
      return sqrt(sd / m);
}
//用平均值和标准差规范化所有样本的特征,col 表示 X 的某个特征对应的某一列
auto Normalization(double X[][n], const int m, const unsigned int col,
double &mean_value, double& sd) {
    mean_value = mean(X, m,col) ;
    sd = standard_deviation(X, m,col,  mean_value);
    for (auto i = 0; i != m; i++) {
        X[i][col] = (X[i][col] - mean_value) / sd;
    }
}
```

现在,对 X 的两个特征用上述的函数进行规范化:

```
double means[n], sds[n];     //n 个不同特征的均值和方差
//对 X 的 col 列对应的某特征进行规范化
auto Normalization(double X[][n], const int m, double means[], double sds[],
     const unsigned int start_col = 1) {
     for (unsigned int col = start_col; col != n; col++)
```

```
            Normalization(X, m, col, means[col], sds[col]);
    }
```

接着可以用梯度下降法或批梯度下降法求解代价函数的最值点：

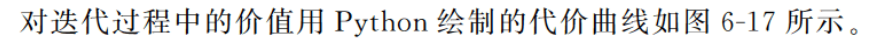
```
int main() {
    //准备训练数据,为数据特征增加一列 1
    double train_X[m][n]{}, train_Y[m]{}, X0[M][n]{},Y0[M];
    get_data(X0, Y0, house_data, M);

    Normalization(X0, M, means, sds);
    double theta[n]{ };                //未知参数
    auto alpha(0.01);
    auto batch_iterations(2);
    auto iterations{ 500 };
    double cost_history[500];
    #if 1                              //所有数据参与的梯度下降法
    gradient_descent(X0, Y0, theta, alpha, iterations,M,cost_history);
    #else                              //部分数据参与的批梯度下降法
    srand(static_cast<unsigned int>(time(0)));
    batch_gradient_descent(train_X, train_Y, X0, Y0, theta, alpha, batch_iterations, iterations,
            M, m, cost_history);
    #endif
    for (int j = 0; j != n; j++)       //输出求得的 theta 值
        cout << theta[j] << '\t';
    cout << '\n';

    for (int j = 0; j != 500; j++)     //输出迭代过程的不同 theta 对应的代价值
        if (j % 39 == 1)
            cout << j << '\t' << cost_history[j] << '\n';
    cout << '\n';
    return 0;
}
```

执行程序，输出结果：

```
338176  103032  -202.325
1       6.42978e+10
40      3.01868e+10
79      1.4965e+10
118     8.07343e+09
157     4.9122e+09
196     3.44199e+09
235     2.74686e+09
274     2.41115e+09
313     2.24447e+09
352     2.15874e+09
391     2.11275e+09
430     2.08688e+09
469     2.07161e+09
```



对迭代过程中的价值用 Python 绘制的代价曲线如图 6-17 所示。

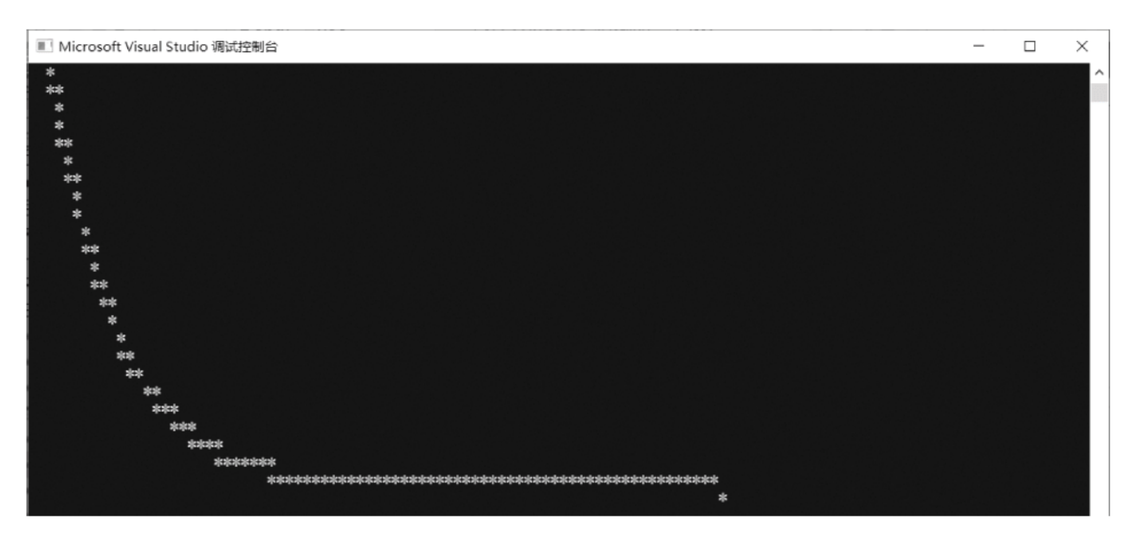


图 6-17 对迭代过程中的价值用 Python 绘制的代价曲线

6.10.10 预测房屋价格

得到了房屋特征值和价格的预测模型，就可以用这个模型去预测一个新的房屋的价格，下面的代码预测一个新房屋（1650m²、房间数是 3）的价格：

```
double pred = x[0] * theta[0] + (x[1] - means[1]) / sds[1] * theta[1]
       + (x[2] - means[2]) / sds[2] * theta[2];
std:cout << pred << endl;
```

即对一个样本，先将相应特征规范化，则带入假设函数 $h_\theta(x)$ 得到该样本对应的目标值。输出的预测值是：

292265

6.11 习题

1. 下列函数定义是否存在错误？为什么？

(1)
```
int what() {
     string s;
     //…
     return s;
 }
```

(2) `hi2(int i) { /* … */ }`

(3) `int name(int v1, int v1) /* … */ }`

(4) `double hello(double x) return x * x;`

2. 函数实参和形参的区别是什么？形参主要分为哪两类？请举例说明。

3. 编写一个函数，当它第 1 次调用时返回 1，第 2 次调用时返回 2，即每次调用时返回

的值都增加 1。

4. 函数的形参、静态局部变量和非静态局部变量有什么区别？编写一个函数，说明它们的区别。

5. 写一个计算 n 的阶乘的函数并测试它。

6. 下面的 2 个函数是否是重定义？

```
void g(int * const p);
void g(const int *p);
```

7. 编写一个函数，判断一个整数是否是一个质数，即只有 1 和自身 2 个因子。

8. 将第 5 章的冒泡排序写成一个如下的函数形式：void sort(int * a,const int n)，并在 main()函数里对一个 int 数组测试这个函数。

9. 歌德巴赫猜想指出：任何一个充分大的偶数都可以表示为两个质数之和。例如：4=2+2,6=3+3,8=3+5,……,50=3+47。编写程序将输入的任意正偶数表示为两个质数之和。

输入：正偶数 n。

输出：n =质数 1+质数 2。

提示：先编写一个判断一个整数是否为质数的函数。

10. 设计一个函数，输入小写英文字母，返回对应的大写英文字母。

提示：所有小写字母的整数值是连续的，如'b'-'a'的值是 1，大写英文字母之间也是一样连续的。

11. 下列 3 个程序的结果是什么？为什么？用堆栈表示 main()函数和 swap()函数的调用关系。

(1)
```
//形参 a,b 的类型是 int
void swap(int a, int b){
    int t = a; a = b; b = t;   //不同语句可以放在一行,只要用分号隔开它们
}
int main(){
    auto x = 3,y = 4;
    swap(x,y);
    std::cout << x <<'\t'<< y << std::endl;
}
```

(2)
```
//形参 a,b 的类型是 int *
void swap(int *a, int *b){
    int t = *a; *a = *b; *b = t;   //不同语句可放在一行,只要用分号隔开它们
}
int main(){
    auto x = 3,y = 4;
    swap(&x,&y);
    std::cout << x <<'\t'<< y << std::endl;
}
```

(3)
```
//形参 a,b 的类型是 int 型引用变量
void swap(int &a, int &b){
    int t = a; a = b; b = t;   //不同语句可以放在一行,只要用分号隔开它们
}
```

```
int main(){
    auto x = 3,y = 4;
    swap(x,y);
    std::cout << x <<'\t'<< y << std::endl;
}
```

12．用冒泡排序法对3个数进行排序和输出，并附带一个默认参数reverse=false，表示默认从小到大排序，当该形参对应的实参为true时，则从大到小排序。

13．模仿一维数组的引用形参，请将6.3.3节的h()函数修改成二维数组的引用形参，并修改二维数组的每个元素为该元素的平方。

14．递归函数是（　　）。

A．在程序中调用所有函数的函数　　B．调用自身的函数

C．调用除自身以外的所有函数的函数　　D．递归函数的形参必须都是整型

15．写一个函数输入一个整数 n，输出 n 行的Pascal三角形，如下所示。

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5   10  10  5   1
```

16．定义如下递归函数Acm(m,n)：

$$\mathrm{Acm}(m,n)=\begin{cases} n+1 & m=0 \\ \mathrm{Acm}(m-1,1) & n=0 \\ \mathrm{Acm}(m-1,\mathrm{Acm}(m,n-1)) & n>0,m>0 \end{cases}$$

其中 m、n 为正整数，并输出Acm(2,1)和Acm(3,2)的值检验是否正确。

17．编写2个同名的area()函数用于根据圆和矩形的参数求圆和矩形的面积，并从键盘输入圆和矩形的参数，调用这2个函数输出对应的圆和矩形的面积。

18．下面程序有没有错误？如有错误，原因是什么？并改正之。

```
#include <iostream>
void print(char const * str) { std::cout << str; }
void print(short num) { std::cout << num; }
int main() {
    print("abc");
    print(0);
    print('A');
}
```

19．运行下列程序的结果是（　　）。

A．5　　B．6　　C．3　　D．语法错误

E．上述说法都不对

```
#include <iostream>
int foo(int x, int y){
```

```
    return x + y;
}
int foo(const int x, const int y){
    return x + y + 1;
}
int main(int argc, char ** argv){
    const int x = 3;
    const int y = 2;
    std::cout << foo(x, y) << std::endl;
    return 0;
}
```

20. 为什么要将函数的形参尽量设成 const 或 const 对象的引用或指针？请结合例子说明。

21. 下面哪组声明是错误的？

(1) `int fun(int, int);`
`int fun(const int, const int);`

(2) `int gun();`
`int * gun();`

(3) `int hun(int * );`
`double hun(double * );`

(4) `int kun(const int);`
`int kun(const int&);`

(5) `int f(int a, int b = 0, int c = 0);`

(6) `char * g(int ht = 24, int wd, char bckgrnd);`

22. 下面哪个调用是错误的？

```
char * f(int ht, int wd = 80, char bckgrnd = ' ');
```

A. f()；　　B. f(24，10)；　　C. f(14，'*')；

23. 有一对兔子，从出生后第 3 个月起每个月都生一对兔子，小兔子长到第 3 个月后每个月又生一对兔子，假如兔子都不死，那么一年后兔子对数为多少？

提示：可用递归函数。

24. 从网上搜索图形学的扫描转换直线段 bresenham 算法，为 ChGL 图形库添加一个绘制线段的 API 函数 plot_line()，并用该函数绘制图 6-15 中的直线。



类 和 对 象

7.1 面向对象编程

一个程序是由数据和对数据处理的指令(语句)组成。传统的**过程式编程**用文字量和变量表示数据,用函数(过程)对这些数据进行处理。对于简单的问题,程序可以用一个 main()主函数对数据进行处理;对于复杂的问题,通常采用分而治之的思想将一个大问题分解为一些更小的问题,对这些小问题,分别用单独的程序块(称为**过程**或**函数**)进行处理。例如前面的游戏编程中,除了游戏的主函数外,还有一些用于初始化数据、处理事件、更新游戏状态(数据)、绘制场景等负责专门功能的函数(过程)。程序中,一个函数可能会调用其他函数,函数之间通过这种相互调用,协作完成一个复杂问题的程序设计任务。

过程式编程中数据和处理数据的过程(函数)是一种松散的关系,也就是说,同样的数据可以被程序中的所有过程(函数)访问,而一个函数也可以访问程序中的不同的数据。

过程式编程这种分而治之解决问题的方法可以处理复杂的问题,使程序结构清晰,提高了程序开发效率、增加了程序的可靠性、可读性和代码的复用性。然而,程序中的数据都是分散的,任何代码都可以访问这些数据,如果数据出现了异常,需要在整个软件系统中查找导致错误的处理代码,使得程序难以有效地跟踪、维护和调试。另外,这些数据都是一些内在数据类型的对象,少量底层的内在数据类型无法直接表示实际问题中的各种丰富概念,如游戏中的各种不同的精灵、员工管理系统的员工、电子商务中的订单等概念。

与过程式编程不同,**面向对象编程**(Object Oriented Programming,OOP)方法就是模拟人类思考解决问题的方法,将一个软件系统看成是由一个个具体对象构成的,每个对象不但包含了这个对象自身的所有信息,还具有自己特有的功能,例如游戏中的每个精灵不但具有位置、大小、图像、生命值等数据属性,还具有运动、更新自身状态等行为能力。面向对象系统中,对象之间通过收发消息协作完成相关的任务,如一个员工向另外一个员工传达一个通知,接到通知的员工就会执行某个动作或行为。

例如,下列代码中表示字符串的 string 类型的变量 str:

```
std::string str("hello world");
```

就是一个具体的**对象**,通过 string 类型的对象 str 可调用 string 类型的 size()方法:

```
std::cout << str.size()<<'\n';
```

size()方法用于查询一个 string 对象的字符个数。上述代码将输出 str 中的字符个数,即 11。str.size()称为向该对象 str 发送了一个消息 size()。

C++通过定义一个类类型描述一个概念,即描述这个概念的所有对象具有的共同特性(包括数据属性和行为属性)。如 string 就是一个描述字符串的类类型,string 类的数据属性表示该类对象包含的字符信息。此外,还有行为或功能属性,如该类的 size()函数用来查询一个字符串对象的字符个数。当然,string 类还有很多其他行为属性(函数)。

面向对象编程通常分为如下 3 步。

首先需要对系统进行分析,识别出系统有哪些种类的对象,即思考系统中有哪些概念。如开发 Pong 游戏程序,需要分析其中有哪些种类的对象,如游戏窗口、画面背景、挡板、球等不同的对象。电子商务系统中有商家、消费者、各种商品、订单、购物车、物流信息等各种概念的对象。除识别出各种概念外,面向对象编程还要分析这些概念之间的关系,如一个员工管理系统中有员工、部门,员工中又分为财务、销售、经理等不同类型的员工。员工和经理概念之间是一种一般到特殊的关系,即经理是一个特殊的员工,经理不但具有一般员工的属性,还有一些自身特有的信息和功能,如经理具有一定的级别、经理能管理一组员工。而部门和员工、经理之间具有一种包含的关系,即一个部门可能包含多个员工和经理。

其次,对于每个概念,还要分析这个概念的所有对象具有哪些共同的属性,属性包含描述对象状态的**数据属性**和描述对象行为能力的**功能属性**。面向对象编程用类描述了这个类的所有对象具有的共同特性(属性),即数据属性(也称**数据成员**或**成员变量**)和功能属性(也称**成员函数**),但为了和普通的全局函数区分,类的成员函数通常被称为**方法**。

最后,需要确定系统中应该有哪些具体的属于不同类的对象,这些对象如何发送和接收消息来协作完成一个任务,并根据类来创建这些对象。

在面向对象编程中,概念之间的关系通常分为 2 种:继承和包含。如《红色警戒》游戏中可以抽象出背景和精灵 2 个概念。背景包含山脉、树木、河流等概念,背景和山脉、树木、河流是一种包含关系。精灵刻画了游戏中所有活动对象的共同属性,如位置、速度、图像等属性,精灵又可以分为建筑、兵种、战车、战机等,它们和精灵之间是一种继承关系,每种物体既具有一般精灵的属性,又具有自己特有的属性,如兵种除继承了一般精灵的属性外,还有生命值、走路、跑步、射击等属性。兵种进一步可以分为步兵、火箭兵、医护兵、工程兵、间谍等。

因此,面向对象程序设计(编程)需要解决下列问题。

(1) 系统中包括哪些概念(类)? 这些概念(类)之间具有的是继承关系还是包含关系?

(2) 每个具体的概念(类)有哪些数据属性(成员变量)和功能属性(方法)?

(3) 系统中包含哪些具体的对象? 这些对象是如何协作完成不同的任务的?

面向对象编程有 3 个特性:封装、继承和多态。

封装：就是用一个类(class)来刻画具有共同特性的类对象，即类描述了该类的对象具有哪些数据属性和行为属性。这些属性中有的是私有的，即外界没法直接访问的，有些是公开的，即外界可以直接访问的。如你遇到一个人，没法直接知道他的年龄、姓名，除非你询问他，他愿意告诉你。将一些属性作为私有的，使得外界不能访问、修改这些私有属性，从而保证了对象数据的完整性和安全性。外界可以通过对象的公开方法去查询对象的信息或请求对象执行某个动作。这些公开方法就是所谓的**接口**。外界只能通过接口访问对象，正如手机的内部电路等被手机外壳封装，外界没法看到其内部，只能通过手机外壳暴露的接口如按键、触摸屏去操作手机，从而保护了手机内部元件的安全。

如图7-1(a)所示，用一个类Person描述"人"这个概念，它刻画了所有人的属性。封装性表达了概念之间的包含关系，如"人"包含了(封装了)"姓名""年龄""性别"以及"说""走""吃"等。

继承：面向对象编程通过从一个已有的类定义一个**派生**类的方式表达概念之间的继承关系。例如，在表示"人"的Person类基础上可以定义表示"学生"和"雇员"的派生类Student和Employee，Person这个类就称为Student和Employee类的**基类(父类、超类)**。反过来，Student和Employee类称为Person类的**派生类(子类)**，如图7-1(b)所示。

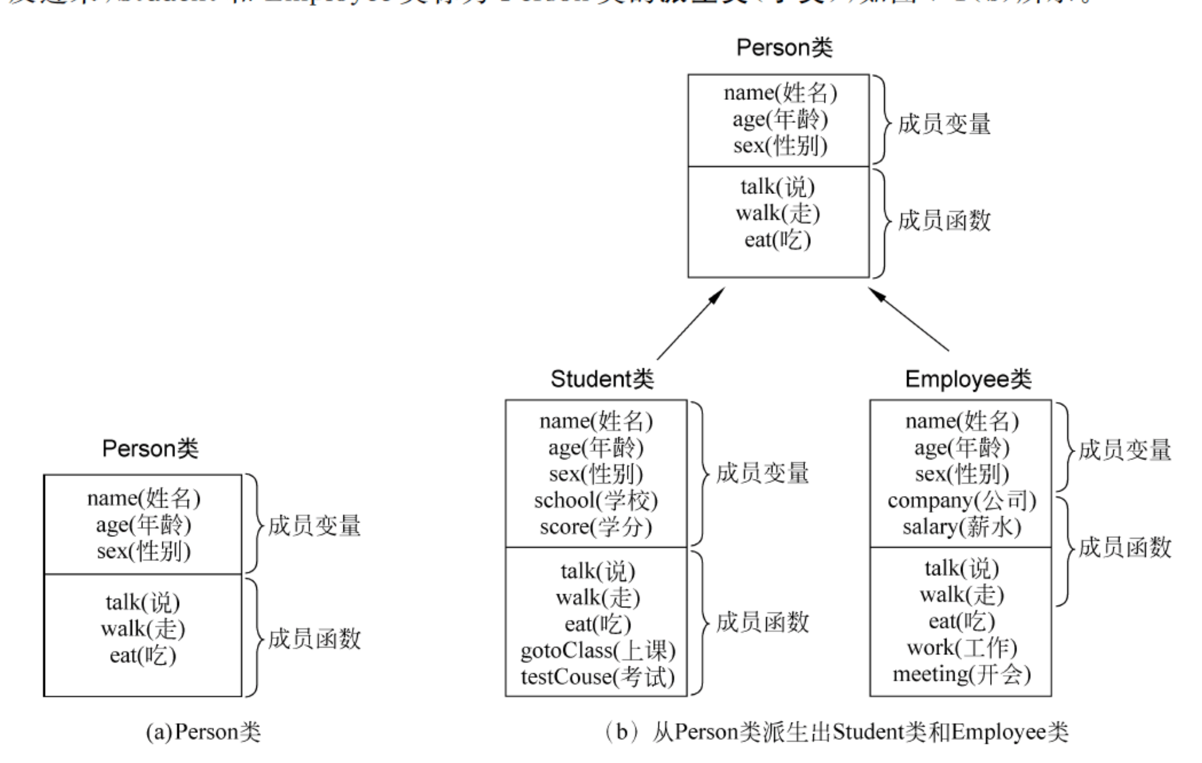


图7-1 Person类及其派生类Student和Employee

多态：一个基类的指针(或引用)可以指向(或引用)派生类的对象，程序运行时会根据基类指针(或引用)实际指向(或引用)的对象的类型而调用这个类型的方法。例如：

```
Person * p{nullptr};
Student s;
Employee e;
p = &s;          //基类 Person 指针变量 p 保存的是派生类 Student 对象 s 的地址
```

```
p->talk();      //调用 p 指向对象的实际类 Student 的 talk()方法
p = &e;         //基类 Person 指针变量 p 保存的是派生类 Employee 对象 e 的地址
p->talk();      //调用 p 指向对象的实际类 Employee 的 talk()方法
```

上述代码中的 p 是一个基类 Person 类型的指针变量，p—> talk()会调用 p 指向的实际对象所属类的 talk()方法而不是基类 Person 的 talk()方法。这种根据基类指针(或引用)指向(或引用)的实际对象的类型而调用该类型的方法的行为，称为**多态**。

面向对象编程和过程式编程是两种不同的分析问题的方法，它们并不矛盾和排斥，实际编程问题可以同时采用这两种编程方法来设计软件系统。C++的函数和类就是分别支持这两种编程思想的具体语言设施。

7.2 类

在 C++中，一个类就是一个用关键字 **class** 或 **struct** 定义的数据类型，称为**用户定义类型**，即程序员自己定义的数据类型。和 C++的内在类型一样，可以定义这些类类型的变量(**对象**)。

一个类定义中可以包含描述类对象状态的变量(称为**成员变量**)和对类对象处理的函数(称为**成员函数**或**方法**)。

7.2.1 定义一个类

用关键字 class 或 struct 定义一个类，其格式如下：

```
class 类名
{
    类的成员...
};
```

或

```
struct 类名
{
    类的成员...
};
```

例如，用 **struct** 关键字定义表示日期的类 Date：

```
//日期类 Date 包括表示年、月、日的成员 year、month、day
struct Date{
    int year{2000},month{1},day{1};
};
```

也可用 class 关键字定义这个 Date 类：

```
class Date{
```

```
    int year{2000},month{1},day{1};
};
```

7.2.2 定义类的对象(变量)

如同内在类型一样,可以定义类的变量(对象)。如下列代码定义了一个 Date 类的变量(对象)day。

```
#include <iostream>
struct Date {
      int year{ 2000 }, month{ 1 }, day{ 1 };
};
int main() {
    Date day;                //day 是 Date 类的变量(对象)
    day.year = 2018;         //通过成员访问运算符.访问类对象 day 的成员 year
    day.month = 6;
    day.day = 1;
    std::cout << day.year << '-' << day.month << '-' << day.day << '\n';
}
```

因为 Date 类有 3 个数据成员 year、month、day,因此,Date 类的对象 day 包含了这 3 个数据成员,可以通过**成员访问运算符.** 访问它的成员变量。如用 day.month 访问 day 对象的 month 成员变量、day.day 访问 day 对象的 day 成员。上述代码通过 3 个赋值语句修改了 day 的这 3 个数据成员的值,然后输出这些数据成员。

运行上述程序的输出是:

```
2018-6-1
```

还可以通过 Date 类型的指针变量访问它指向的 Date 对象:

```
#include <iostream>
struct Date {
    int year{ 2000 }, month{ 1 }, day{ 1 };
};

void print(Date * date) {    //date 是 Date * 指针类型,不是一个 Date 对象
    //间接访问运算符->访问指针变量 date 指向的 Date 对象的成员 year,month 和 day
    std::cout << date->year << "-" << date->month << "-" << date->day << '\n';
}

int main() {
    Date day;
    day.year = 2018;         //通过成员访问运算符.访问类对象 day 的成员 year
    day.month = 6;
    day.day = 1;
    print(&day);             //将 day 的地址作为指针传给 print()函数
}
```

print()函数中,date 是一个 Date * 类型的指针变量,程序通过**间接访问运算符**->访问指针变量 date 指向的那个 Date 对象的成员变量,如 date->year。而 main()函数中将 Date 对象 day 的地址(&date)传递给 print()函数的 date 指针变量。

可以用 **class** 关键字代替 struct 关键字定义 Date 类。

```
#include <iostream>
class Date {
    int year{ 2000 }, month{ 1 }, day{ 1 };
};
int main() {
    Date day;
    day.year = 2018;        //通过成员访问运算符.访问类对象 day 的成员 year
    day.month = 6;
    day.day = 1;
}
```

但编译器报告错误:

```
1>f:\7.cpp(53): error C2248: "Date::year": 无法访问 private 成员(在"Date"类中声明)
…
1>f:\7.cpp(54): error C2248: "Date::month": 无法访问 private 成员(在"Date"类中声明)
…
1>f:\7.cpp(55): error C2248: "Date::day": 无法访问 private 成员(在"Date"类中声明)
…
```

即不能通过**成员访问运算符.**访问对象 day 的成员 year(如 day.year)。这是因为用 class 关键字定义的类的成员默认都是 **private(私有的)**,即除了类自身的成员函数外,外部函数如 main()不能访问类对象的 private 成员,而 struct 定义的类成员默认都是 **public(公开的)**,外部函数可以访问类对象的这些公开成员。

可以通过在定义类成员前添加**访问控制修饰符**如 **private** 或 **public** 来修改这些成员是否对外界公开或私有,如:

```
#include <iostream>
class Date {
public:                          //其后的成员声明都是公开的(public)
    int year{ 2000 }, month{ 1 }, day{ 1 };
};
void print(Date * date) {
    std::cout << date->year << "-" << date->month << "-" << date->day << '\n';
}
int main() {
    Date day;
    day.year = 2018;        //通过成员访问运算符.访问类对象 day 的成员 year
    day.month = 6;
    day.day = 1;
    print(&day);
}
```

Date 类的 public 修饰符表示其后面的成员是公开的，即 Date 对象(如 day)的 3 个成员变量现在都是可以被外界访问、修改的。再编译这个程序，就没有任何编译错误了。

因此，struct 和 class 关键字都可以用来定义一个类，其唯一区别是 struct 定义类的成员默认是 public 的，而 class 定义类的成员默认是 private 的。struct 主要是为了使 C++向后兼容 C 语言中的 struct 结构类型，今后 C++程序中应尽量用 class 定义类，并可通过访问控制修饰符控制类对象的成员能否被外界直接访问。

7.2.3 成员函数

前面 Date 类中的成员都是一些变量即数据，类中还可以包含函数，即**成员函数**(也称为**方法**)。例如，可以在 Date 中定义一个 print()成员函数：

```
#include <iostream>
class Date{
    public:                    //public 之后的成员都是公开的
    int year{2000},month{1},day{1};
    void print(){
        //print()函数知道 year、month、day 是哪个对象的成员
        std::cout << year <<" - " << month <<" - " << day <<'\n';
    }
};
int main(){
    Date day,day2;
    day.year = 2018;
    day.month = 6;
    day.year = 1;
    day.print();               //必须通过类对象 day 调用类 Date 的函数成员 print()
    day2.print()
}
```

day.print()通过类对象 day 调用类 Date 的函数成员 print()，该方法就会输出这个 day 对象的 year、month、day 成员变量。

上述程序的输出是：

```
2018 - 6 - 1
2010 - 1 - 1
```

7.2.4 this 指针

类 Date 的成员函数 print()必须通过一个 Date 类对象如 day 去调用 day.print()。编译器实际上会将类的成员函数转换为普通的函数，即类似 void print(Date * this)形式的普通函数。

类的成员函数都会被编译器转换为一个普通的全局函数，这个普通函数包含一个特殊的叫作 this 的指针形参，这个形参就指向调用这个函数的那个对象(如 day)，即存储这个调

用对象(day)的地址。

通过类对象调用成员函数如执行 day. print()时，会将成员函数的调用转换为普通的函数调用，如 day. print()被转换为 print(&day)，day 的地址就被传给编译器生成的普通函数 void print(Date * this)的 this 形参。因此，在成员函数中访问对象的数据成员就是通过这个隐含的 this 指针去访问的。

对于 Date 类，编译器会将 print()成员函数转换为下面形式的普通函数。

```
void print(Date * this) {
    //通过指针变量 this 访问其指向对象的 year、month、day 成员
    std::cout << this -> year << " - " << this -> month << " - " << this -> day << '\n';
}
```

即通过 this—> year 访问 this 指向对象的 year 数据成员。这就是为什么“必须通过一个类对象去调用类的非静态成员函数”的原因(**注**：后续章节会介绍类的静态和非静态成员函数)。

因此，在类的非静态成员函数中可以使用 this 指针(但函数的参数里不能声明 this 指针)。

```
class Date {
public:                         //public 之后的成员都是公开的
    int year{ 2000 }, month{ 1 }, day{ 1 };
    void print()                //注意,不能写成: void print(Date * this)
    {
        //通过指针变量 this 访问其指向对象的 year、month、day 成员
        std::cout << this -> year << " - " << this -> month << " - " << this -> day << '\n';
    }
};
```

可以让一个类的非静态成员函数返回这个 this 指针或这个 this 指针指向的对象的引用。如下面的 setDay()和 setYear()成员函数都返回这个 this 指向对象的引用。

```
#include < iostream >
class Date {
public:                         //public 之后的成员都是公开的
    int year{ 2000 }, month{ 1 }, day{ 1 };
    void print() { std::cout << year << " - " << month << " - " << day << '\n'; }
    Date& setDay(int d) {
        day = d;
        return * this;          //返回 this 指针指向的对象的引用
    }
    Date& setYear(int y) {
        year = y;
        return * this;          //返回 this 指针指向的对象的引用
    }
};
int main() {
    Date day;
```

```
    day.year = 2018;
    day.month = 6;
    day.day = 1;
    day.print();
    day.setYear(2019).setDay(20);     //day.setYear(2019)的返回是 day 自身的引用
                                  //因此对它可以继续调用 setDay(20)
}
```

day.setYear(2019)的返回是 day 自身的引用，因此对它可以继续调用 setDay(20)，即可以将返回**自引用**的方法串起来使用。

当然，也可以让类的普通成员函数返回这个 this 指针，如下面的 setYear()成员函数。

```
#include <iostream>
class Date {
public:                           //public 之后的成员都是公开的
    int year{ 2000 }, month{ 1 }, day{ 1 };
    void print() { std::cout << year << "-" << month << "-" << day << '\n'; }
    Date& setDay(int d) {
        day = d;
        return *this;             //返回 this 指针指向的对象的引用
    }
    Date* setYear(int y) {
        year = y;
        return this;              //返回 this 指针
    }
};
int main() {
    Date day,day2;
    day.year = 2018;
    day.month = 6;
    day.day = 1;
    day.print();
    day2.setYear(2019)->setDay(20);     //day.setYear(2019)的返回是 day 的指针
                                  //通过这个指针可以间接调用 setDay(20)
}
```

Date 的 setYear()返回的是调用这个方法的对象的指针，因此可以通过这个指针间接访问这个对象的属性(包括通过指针间接调用类的成员函数)。

程序运行结果：

```
2018-6-1
2000-1-1
```

每个对象的数据成员都有自己单独的内存，但类的成员函数为类的所有对象共享，如图 7-2所示。编译器将类的成员函数转换为普通的外部函数，其中包含了一个指针变量，指向调用这个函数的类对象，通过类对象调用成员函数，就是将这个对象的指针传递给这个函

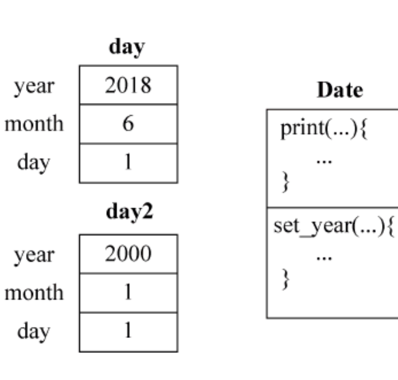


图 7-2 每个对象占据独立的内存

数。即成员函数的代码只占据一块内存，是所有对象共享的，而每个对象的数据成员要占据独立的内存。

7.2.5　类对象的大小

一个类对象占据的内存存放的是其数据成员，因此类对象的大小基本上等于或略大于所有数据成员占据内存之和。

为什么略大于所有数据成员之和呢？这是因为数据在内存里是要对齐存放的，一个变量如果是 2 字节，则其地址是 2 的倍数，如果是 4 字节，则其地址是 4 的倍数。如果一个类中前 3 个成员都是 4 字节，还有 1 个 8 字节的成员，则会按 8 字节对齐，即在最后 8 字节的前面额外多分配 4 字节的空闲内存。即一共占据 4+4+4+4+8=24 字节。

和内在类型一样，可以用 sizeof()运算符检查一个类对象占据内存的大小，传入一个具体的类对象或者类本身。例如：

```
#include <iostream>
class X {
    int a, b, c;
    double d;
};
int main() {
    X x;
    std::cout << sizeof(3) << '\t'<< sizeof(double) << '\n';
    std::cout << sizeof(x) << '\t'<< sizeof(X) << '\n';
}
```

程序运行结果：

```
4       8
24      24
```

这也说明了类对象的内存中只有数据成员，没有类的成员函数代码。因为成员函数都会被编译器改造成普通的函数，这些函数代码占据另外的单独内存而不是放在每个对象自身的内存中。

7.3　构造函数

7.3.1　创建类对象的构造函数

在定义类 Date 的成员 year、month、day 时，可以给它们一个初始化的默认值，也就是所有 Date 类对象的这些数据成员都将具有同样的默认值。

```
#include <iostream>
class Date {
    int year{ 2000 }, month{ 1 }, day{ 1 };
```

```
public:
    void print() { std::cout << year << "-" << month << "-" << day << '\n'; }
};
int main() {
    Date day, day1;
    day.print();
    day1.print();
}
```

程序运行结果：

```
2000-1-1
2000-1-1
```

这个 Date 类的所有对象(如 day 和 day1)的数据成员是完全一样的。一般地，类的不同对象的数据成员的值是不同的。对于上面的 Date 类，因为 Date 类对象的数据成员是私有的，外部函数也没法修改这些对象的数据成员。

如何在定义类对象时用不同的数据初始化对象的数据成员呢？

实际上，在定义类对象的时候，编译器会自动调用一个特殊的叫作**构造函数**的成员函数对类对象的数据成员初始化。构造函数是函数名和类名相同且没有返回类型的类成员函数，如果程序员没有定义构造函数，则编译器会自动生成一个参数列表和函数体都为空的**默认构造函数**。即 Date 类代码实际如下：

```
#include <iostream>
class Date {
    int year{ 2000 }, month{ 1 }, day{ 1 };
public:
    Date() {}                  //构造函数名与类名相同，没有返回类型
                               //默认构造函数没有任何参数和函数体代码
    void print() { std::cout << year << "-" << month << "-" << day << '\n'; }
};
```

其中，成员函数 Date()是一个构造函数，这种不带参数(或所有参数都具有默认值)的构造函数叫作**默认构造函数**。

为了验证定义类对象是否会自动调用构造函数，可以在构造函数里添加一些输出语句。

```
#include <iostream>
class Date {
    int year{ 2000 }, month{ 1 }, day{ 1 };
public:
    Date() { std::cout << "构造 Date 对象：" << std::endl; print(); }
    void print() { std::cout << year << "-" << month << "-" << day << '\n'; }
};
int main() {
    Date day;                  //定义类对象时自动调用匹配的构造函数
}
```

定义 Date 对象 day 时会自动调用默认构造函数 Date()。执行该程序，屏幕上将显示：

```
构造 Date 对象:
2000 - 1 - 1
```

即先执行第一句输出，然后调用 print()成员函数。

也可以定义带输入参数的构造函数，在定义类对象时用输入的实参对类对象初始化：

```
#include <iostream>
class Date {
    int year{ 2000 }, month{ 1 }, day{ 1 };
public:
    Date(int y, int m, int d) {
        year = y; month = m; day = d;
        std::cout << "构造 Date 对象: " << '\t';
        print();
    }
    void print() { std::cout << year << " - " << month << " - " << day << '\n'; }
};
int main() {
    Date day(2018,8,18),    //定义类对象时自动调用匹配的构造函数
        day2{ 2019,6,1 };
}
```

程行运行结果：

```
构造 Date 对象:    2018 - 8 - 18
构造 Date 对象:    2019 - 6 - 1
```

构造函数带有 3 个形参，在定义类对象时也必须提供对应的实参。可以采用函数调用形式即圆括号传递实参，也可以用{}形式的列表初始化提供实参。

和普通的函数调用一样，如果定义类对象时少于或多于 3 个实参，则编译器会报错。

```
Date day;                   //错: 缺少默认构造函数
Date day(2010, 1, 2, 3);    //错: 没有重载函数接受 4 个参数
```

"Date day;"的"缺少默认构造函数"的错误是因为一旦定义了自己的构造函数，编译器就不再生成默认构造函数了。如果仍然想使用默认构造函数，怎么办？可以添加下列不带参数的默认构造函数

```
Date() {}
```

也可以通过 **default** 关键字来通知编译器生成默认构造函数。即将上面的语句改为：

```
Date() = default;
```

即定义如下的包含 2 个构造函数的 Date 类：

```
#include <iostream>
class Date {
        int year{ 2000 }, month{ 1 }, day{ 1 };
public:
        Date() = default;
        Date(int y, int m, int d) {
             year = y; month = m; day = d;
             std::cout << "构造 Date 对象: " << '\t';
             print();
        }
        void print() { std::cout << year << "-" << month << "-" << day << '\n'; }
};
int main() {
        Date day(2018, 8, 18),//定义类对象时自动调用匹配的构造函数
             day2{ 2019,6,1 };
        Date day;            //ok,调用默认构造函数
}
```

和普通函数一样,类的成员函数(包括构造函数)的参数也可以有默认值,并遵守默认参数一律靠右的规则:

```
class Date {
    int year{ 2000 }, month{ 1 }, day{ 1 };
public:
    Date(int y = 2000, int m = 1, int d = 1) {
          year = y; month = m; day = d;
    }
    void print() { std::cout << year << "-" << month << "-" << day << '\n'; }
};
int main() {
    Date day, day1(2011), day2{ 2019,6},
          day3{ 2019,13,8 };
}
```

可以提供不同个数的实参调用上面的构造函数 Date(int y = 2000, int m = 1, int d = 1)初始化类对象。

因为这个构造函数的每个参数都有默认值,所以它就是默认构造函数。因为有了这个默认构造函数,所以不能在该类中再添加不带参数的默认构造函数。如:

```
class Date {
    int year{ 2000 }, month{ 1 }, day{ 1 };
public:
    Date() = default;
    Date(int y = 2000, int m = 1, int d = 1) {
         year = y; month = m; day = d;
    }
    void print() { std::cout << year << "-" << month << "-" << day << '\n'; }
};
```

编译器编译上面代码时会报告“重定义”的错误,因为定义了 2 个默认构造函数。

7.3.2 初始化成员列表

对于构造函数，可以在函数体前面对类的数据成员进行初始化。如：

```
class Date {
    int year{ 2000 }, month{ 1 }, day{ 1 };
public:
    Date(int y = 2000, int m = 1, int d = 1) :year{y},month(m),day(d){       }
    void print() { std::cout << year << "-" << month << "-" << day << '\n'; }
};
```

即在构造函数头后，函数体前，以冒号:开头，用**成员名{形参名}**或**成员名(形参名)**形式对每个成员初始化，并以逗号隔开。这种初始化类对象成员的方式称为**初始化成员列表**。其好处是提高程序执行效率，避免了“在进入构造函数前先默认初始化类成员，然后在构造函数体里再对这些成员重新赋值”，而直接用构造函数的参数对类对象的成员初始化一次，函数体中不再重新初始化。

注意：使用初始化成员列表对类对象的数据成员初始化时，是按照这些数据成员在类中出现的次序初始化的，和它们在初始化列表中出现的次序无关。如果将构造函数写成如下形式：

```
Date(int y = 2000, int m = 1, int d = 1) : day(d), month(m), year{y}{}
```

构造函数仍然按照数据成员在 Date 中定义的次序，即 year、month、day 的次序，依次初始化。

7.3.3 拷贝构造函数

可以定义一个类对象并用同一个类的其他对象初始化，假如有一个类 X 的对象 x，则可以用 x 去初始化一个新的 X 类对象：

```
X y{x};  //也可以写成 X y(x);
```

即定义了 X 的对象 y，并用 X 的已有对象 x 对 y 进行初始化构造。例如：

```
int main() {
    Date day{ 2018,1,1 },day2{day};
    day.print();
    day2.print();
}
```

将输出如下结果：

```
2018-1-1
2018-1-1
```

day 和 day2 对象具有完全一样的数据成员值。在定义 day2 对象时传递的是 day 对

象，产生的 day2 对象是 day 对象的复制(拷贝)。

但前面的 Date 类中并没有 Date 对象作为参数的构造函数，为什么没有报告编译错误？这是因为定义 day2 对象时调用的是一种称为**拷贝构造函数**的特殊构造函数，如果程序员在自定义类中没有定义拷贝构造函数，编译器会自动生成一个默认的拷贝构造函数。默认拷贝构造函数会用一个已有的类对象构造一个内存内容完全一样的新的类对象。即 2 个对象的内存的二进制位都是一模一样的，这种拷贝称为**硬拷贝**。

对于普通的类，默认拷贝构造函数的这种硬拷贝是没任何问题的，但如果一个对象需要使用一些资源如动态内存、打开文件或网络端口，这种硬拷贝会使得 2 个对象共享同一个资源，硬拷贝会导致灾难性后果。

对于一个类 X，拷贝构造函数的函数规范是 X(const X&x)，即其参数是一个该类的 const 对象的引用。对于上面的 Date 类，编译器默认生成的拷贝构造函数是：

```
Date(const Date& d) :year{ d.year }, month(d.month), day(d.day){}
```

即每个数据成员逐一硬拷贝。为了验证定义类对象 day2{day}时，是否调用了拷贝构造函数，可以自己定义这个拷贝构造函数并添加一些打印语句：

```
Date(const Date& d) :year{ d.year }, month(d.month), day(d.day){
            std::cout << "拷贝构造函数\n"; print();
}
```

执行程序，输出结果：

```
拷贝构造函数
2018 - 1 - 1
2018 - 1 - 1
2018 - 1 - 1
```

前 2 行是拷贝构造函数的输出信息。

拷贝构造函数中的形参就是类对象的引用，能否是值参数呢？即能否将拷贝构造函数写成下面的形式？

```
X(X x);
```

答案是不行的。假如定义 Date 类的拷贝构造函数如下：

```
Date(Date d) :year(d.year),month(d.month),day(d.day){}
```

当调用这个拷贝构造函数时，需要将已有对象(如 day)作为实参初始化给这个构造函数的形参 d，即执行 Date d{day}，这就又调用这个拷贝构造函数，如此就会无限循环下去了。

因此，拷贝构造函数的形参至少应该是引用参数：

```
Date(Date &d) :year(d.year),month(d.month),day(d.day){}
```

这样，形参直接引用实参，参数传递时不存在复制的问题，解决了无限循环问题。

然而这个构造函数还有一个问题，就是 const Date 类型的对象不能作为该函数的实参(因为 non-const 对象的引用不能用 const 对象初始化)。但拷贝构造函数只是用来创建新的对象，不会修改已有的对象，为什么不将参数设置成 const 对象的引用(refernece to const)呢？如：

```
Date(const Date &d) :year(d.year),month(d.month),day(d.day){}
```

这样定义的拷贝构造函数既避免了无限循环，又可以接受 const 和 non-const 对象作为实参。因此，对于一个类 X，其拷贝构造函数的函数规范是：

```
X (const X& x);
```

7.3.4　赋值运算符：operator＝

默认情况下，可以将一个类对象赋值给另外一个类对象，如：

```
Date day{ 2018,1,1 }, day3;
day3 = day;
day.print();
day3.print();
```

程序输出：

```
2018 - 1 - 1
2018 - 1 - 1
```

通过赋值运算 day3 ＝ day，day3 对象复制了 day 对象，即两者的数据成员值是一样的，赋值运算符和拷贝构造新对象复制已有对象的区别是，赋值运算符是在 2 个已经存在的对象之间的复制(拷贝)，而拷贝构造函数是用已有对象创建一个新对象。

为什么对类的对象能用赋值运算符＝？因为对于一个类，编译器会生成一个默认的赋值运算符函数，即 **operator＝**()成员函数，对于类类型 X，其格式类似于拷贝构造函数：

```
X&   operator = (const X &object);
```

上述代码形参也是一个 const 对象的引用，但其返回类型是**对象的引用**，即被赋值的对象自身。**operator＝**是赋值运算符＝的完整函数名。

如上面的 day3＝day 就是调用了编译器为 Date 类默认生成的赋值运算符函数 **operator＝**()。即

```
Date&   operator = (const Date &object);
```

默认的赋值运算符函数也是和默认拷贝构造函数一样，将一个对象的数据硬拷贝到另一个对象中。对于不占用资源的类来说，这没有任何问题；对于占用资源的类，和拷贝构造函数一样，程序员应该重新定义赋值运算符函数。

下面代码对 Date 类重新定义了赋值运算符函数 operator=()，当然执行的仍然是硬拷贝操作，对这个不占用资源的类来说，没有任何问题。

```
Date &operator = (const Date& date) {
    std::cout << "赋值运算符\n";
    this->year = date.year;
    this->month = date.month;
    this->day = date.day;
    return *this;           //返回被赋值对象的而自身的引用
}
```

因为赋值运算符函数返回的是被赋值对象自身的引用(即自身)，因此，可以将赋值运算符串起来使用，如下面代码中的 day3=day2=day。

```
int main() {
    Date day(2018, 1, 1);
    Date day2, day3;
    day3 = day2 = day;      //先执行 day2 = day,结果是 day2,再执行 day3 = day2
                            //day2 = day 实际是 day2.operator=(day)
}
```

上述程序的输出是：

```
赋值运算符
赋值运算符
```

上述代码中，day2 = day 实际是 day2.**operator=**(day)的简化写法。

注意：赋值运算符是**右结合性**，即从右往左执行赋值运算符的。先执行 day2=day，然后再执行 day3=day2。最后该语句 day3=day2=day 返回的是 day3 的自引用。

7.3.5 隐式类型转换、explicit

1. 隐式类型转换

带有一个参数的构造函数，定义了**隐式类型转换**，如：

```
#include <iostream>
class A {
public:
    A(int x) { std::cout << "用" << x << "构造对象\n"; }
};
```

构造函数 A(int x)接受一个 int 类型参数，创建一个 A 类的对象。因此，可以如下创建 A 类的对象：

```
A a(1),b{2},c = 3;          //创建 A 类的 3 个对象 a,b,c
a = 4;                      //将 4 赋值给 A 类对象,先将 4 转换为 A 类对象 A(4),再赋值给 a
```

可以看到在初始化一个 A 类对象或对一个 A 类对象赋值时，就是将 int 类型的值转换为 A

类的值。即构造函数 A(int x)实际上定义了从 int 到 A 的隐式类型转换。再看下面的代码。

```
void f(A a) {}
int main() {
    f(2);                    //隐含通过构造函数 A(int)转换成 A 类型对象,再对 a 初始化
}
```

函数 f()接受一个 A 类型的对象作为参数,而函数调用 f(2)中的实参是 int 类型的值 2,用实参 2 对形参 a 初始化时就会调用 A 的构造函数 A(int x)将 int 类型的 2 转换为 A 类的值,再赋值给对象 a。

```
f(A(2));
```

因此,只要在需要 A 对象的地方传递的是 int 类型的值,都会自动将 int 类型的值转换为 A 类型对象。即带有一个参数的构造函数实际上定义了一个隐式类型转换,可以自动将参数类型的值转换为这个类类型的值。

2. 用 explicit 禁止隐含类型转换

然而,对于有的类,这种隐含类型转换会带来一些问题。

```
#include <iostream>
class Circle{
    double radius{0.};
public:
    Circle(double r):radius(r) {}
    auto area() { return 3.1415 * radius * radius; }
    auto isAreaLargerThan(Circle c) { return area() > c.area(); }
};
```

Circle 构造函数定义了从 double 到 Circle 类型的隐含类型转换。对于下面代码:

```
int main() {
    Circle c(2), c2(5);
    std::cout << "c 和 c2 的面积是: " << c.area() << '\t' << c2.area() << '\n';
    if (c2.isAreaLargerThan(c))
        std::cout << "c2 的面积比 c 大\n";
    else
        std::cout << "c2 的面积比 c 小\n";
    if (c2.isAreaLargerThan(50))
        std::cout << "c2 的面积比 50 大\n";
    else
        std::cout << "c2 的面积比 50 小\n";
}
```

c2.isAreaLargerThan(50)调用成员函数 isAreaLargerThan(Circle c),该函数接受的是 Circle 类型的对象,而传递的 50 是 int 类型,就会先将 int 类型转换为 double 类型,再调用构造函数 Circle(double)执行自动的隐式类型转换,将 50 转换为 Circle 对象,即构造一个半径是 50 的 Circle 对象。c2.isAreaLargerThan(50)的本来意图是和面积是 50 的圆比较而

不是和半径是50的圆比较，由于这种隐式类型转换，现在变成和半径是50的圆比较面积大小，导致程序运行结果不符合预期的结果：

```
c 和 c2 的面积是: 12.566   78.5375
c2 的面积比 c 大
c2 的面积比 50 小
```

为了防止误将一个整型或浮点数传给这个带一个参数的Circle构造函数，可以在编写这个类时在该构造函数名前添加关键字**explicit**，禁止隐含调用这个构造函数，即添加了该关键字的构造函数只能显式调用。如：

```
explicit  Circle(double r):radius(r) {}
```

改写了这个构造函数后，代码c2.isAreaLargerThan(50))就会报错：

```
...error C2664: "bool Circle::isAreaLargerThan(Circle)": 无法将参数 1 从"int"转换为"Circle"
1>   f: \7.cpp(289): note: class"Circle"的构造函数声明为"explicit"
```

从而有助于程序员早期发现这个隐含的错误。

7.3.6 委托构造函数

一个类可以有多个构造函数，提供不同方式创建类对象。一个构造函数可以调用其他的构造函数，从而可以避免重复的代码。如：

```
#include <iostream>
class Date {
    int year{ 2000 }, month{ 1 }, day{ 1 };
public:
    Date(int y = 2000, int m = 1, int d = 1):year(y), month(m), day(d) {
    }
     //委托 Date(int y = 2000, int m = 1, int d = 1)构造函数
    Date(int *p) :Date(p[0], p[1], p[2]) {  }
    void print() { std::cout << year << "-" << month << "-" << day << '\n'; }
};
```

Date类除接受3个int类型参数的默认构造函数外，还接受一个int *指针参数的构造函数，这个构造函数通过调用3个参数的构造函数将构造对象的工作委托给3个参数的构造函数。Date(int *p)称为**委托构造函数**，而Date(int y = 2000, int m = 1, int d = 1)称为**被委托构造函数**。

执行下列程序：

```
int main() {
    int date[]{ 2018,9,6 };
    Date d(date); d.print();
}
```

将输出：

```
2018 - 9 - 6
```

注意：

- 委托构造函数只能在初始化成员列表里而不能在函数体里调用被委托构造函数。
- 成员变量不能在委托构造函数的初始化列表里初始化，但可以在函数体里初始化成员变量。当然如果在函数体里初始化成员变量，是否有必要将该函数定义为委托构造函数就值得推敲一下了。下面的代码：

```
Date(int *p) :Date(p[0], p[1], p[2]),day{20} {}
```

是错误的。

7.3.7　delete

有时，可能希望禁止某个构造函数或赋值运算符，如禁止编译器生成默认的拷贝构造函数或赋值运算符，可以通过 delete 关键字显式地进行说明。如：

```
class A {
public:
    A(int) {/ * ... * / }
    A(double) = delete;
    A& operator = (const A& o) = delete;
    A(const A& o) = delete;
};
int main() {
    A a(1);
    A a1(3.14);              //错: A(double)被禁止了
    A a2(a);                 //错: A(const A& o) 被禁止了
    a2 = a1;                 //错: A& operator = (const A& o) 被禁止了
}
```

A 的带 double 参数的构造函数、拷贝构造函数和赋值运算符函数都被禁止了，即不能使用这些函数创建对象或对对象赋值。

输入输出流对象也是禁止被复制的，因为代表输入的键盘和输出的屏幕只有一个，如果允许复制，就混乱了。

7.3.8　类对象数组

对于一个没有构造函数或者定义了默认构造函数的类，可以定义这种类的对象的数组。

```
#include <iostream>
class X {
    int x{ 0 };
public:
```

```
    void set_x(int i) { x = i; }
    void print() { std::cout << x; }
};
int main() {
    X x;
    x.print(); std::cout << '\n';
    X arr[3];                   //X 类对象的数组
    for (auto x : arr) {
        x.print(); std::cout << '\t';
    }
    std::cout << '\n';
    for (auto i = 0; i != 3; i++) arr[i].set_x(2 * i + 1);
    for (auto x : arr) {
        x.print(); std::cout << '\t';
    }
    std::cout << '\n';
}
```

程序运行结果：

```
0
0       0       0
1       3       5
```

在定义一个类对象的数组时，会自动调用默认构造函数对数组的每个对象进行初始化创建。上面的代码定义“X arr[3];”的结果是每个数组元素 arr[i]作为一个 X 类的对象，都调用了默认构造函数，其数据成员 arr[i].x 的值都初始化为 0。

对于一个定义了构造函数但没有默认构造函数的类，则不能定义类对象的数组，因为对数组元素的每个对象初始化创建时需要调用构造函数，构造函数需要传递一个参数，而定义数组则没法给每个数组元素传递这些参数。如：

```
class X {
    int x{ 0 };
public:
    X(int i) :x(i) {}
};
int main() {
    X arr[3];                   //X 没有默认构造函数,因此不能定义 X 类型的数组
}
```

因为 X 没有默认构造函数，在对 arr 的每个元素 arr[i]初始化时，其构造函数需要一个参数，但却无法提供。因此，编译会报告错误：

```
error C2512: "X":没有合适的默认构造函数可用
```

7.3.9 类体外定义成员函数和构造函数

一个类的成员函数也可以在类体外定义，但必须在函数名前添加类的作用域，即**类名::**，

用于说明这个函数不是普通的全局函数(外部函数)而是属于一个类的成员函数。如:

```
#include <iostream>
class Date {
    int year{ 2000 }, month{ 1 }, day{ 1 };
public:
    Date(int y = 2000, int m = 1, int d = 1);
    void print();
};
Date::Date(int y, int m, int d) :year(y), month(m), day(d) {}
void Date::print() {std::cout << year << "-" << month << "-" << day << '\n'; }
```

上述的 Date()和 print()成员函数在类外定义,必须有类的作用域作为前缀,如 **Date::** print()。

尽管在类体外定义成员函数,但在类体内仍必须声明该函数。另外,默认参数只能在类体内的函数声明处说明,类体外的函数定义不能再有默认值。如 Date 构造函数的类体外定义中,其形参不能有默认值,默认值只能在类内部的函数声明中说明。

一个良好的习惯,就是通常将类定义的代码放在一个如 X.h 的头文件中,而类的成员函数定义在类体外,如另外一个 X.cpp 文件中,其他源文件如 other.cpp 中如果需要用到这个类 X,只需要包含(#include)这个头文件 X.h 就可以了,但不能包含 X.cpp 源文件,否则就是重定义,违背了 **ODR**(一次定义规则)。这种类成员函数的声明和定义分开还有一个好处,就是如果该类作为库的一部分给第三方使用时,只需要提供头文件和目标文件,而不需要提供具体实现的源代码,从而保护了作者的知识产权。

练习:请将 Date 类的声明放在一个头文件 Date.h 中,而其所有成员函数的实现都放在另外的 Date.cpp 中,然后编写一个 test.cpp,在 test.cpp 中定义个 main()函数,在该函数中,定义 Date 类的对象。注意,Date.cpp 和 test.cpp 中都必须包含 Date.h 头文件,即 #include "Date.h"。另外需要说明的是,用#include 包含一个不在系统路径下的文件 xxx 时,应该用双引号而不是左右箭头,即用#include "xxx"而不是#include <xxx>,表示除系统路径外,还会在当前目录中寻找该文件。

7.4 访问控制和接口

一个好的类实现应该隐藏保护其关键信息,只提供少量的公开成员作为对外的接口。如前面的 Date 类的 3 个数据成员都是 private 的,从而防止一个对象的这些数据被外部代码随意修改或破坏,保证了数据的完整性和安全性。

然而,前面的 Date 类对象只有定义时的初始化,以及通过 print()打印其信息,外界无法查询或修改该对象的单个数据成员。设想一个银行账户的信息永远是创建时的状态,无法存取款、修改账户信息,这是不现实的。因此,Date 类的 3 个数据成员虽然设置成私有的,还应该为经过认证的用户或访问者提供查询或修改其数据成员(如年份)的功能。这通常是通过定义一些公开的(public)成员函数来实现的。如:

```
#include <iostream>
class Date {
```

```
public:
    Date(int y = 2000, int m = 1, int d = 1):year(y), month(m), day(d) {
    }
    Date(int *p) :Date(p[0], p[1], p[2]) {      //委托 Date(int y = 2000, int m = 1, int d = 1)
                                                //构造函数
    }
    //get 函数: 访问类对象的数据
    int getYear() { return year; }
    int getMonth() { return month; }
    int getDay() { return day; }

    //set 函数: 修改类对象的数据
    void setYear(int y) { if (y > 0) year = y; }
    void setMonth(int m) { if (m > 0) month = m; }
    void setDay(int d) { if (d > 0) day = d; }
    void print() { std::cout << year << "-" << month << "-" << day << '\n'; }
private:
    int year{ 2000 }, month{ 1 }, day{ 1 };
};
int main() {
    Date day;
    day.setYear(2018);
    std::cout << day.getMonth() << '\n';
}
```

上述代码中 public 关键字后的成员都是公开的，而 private 关键字后的成员都是私有的，一个对象的私有成员只有该类的成员函数才能访问，即 private 修饰的成员被隐藏起来，外部函数无法看到对象的这些私有成员，而 public 修饰的是公开的成员，外部代码可以访问，所有 public 成员称为这个类的对外的接口。

通过公开的成员函数如 getYear()、setYear()，外部函数可以查询、修改 Date 类对象的私有数据成员。类的成员函数能保证类对象数据的安全性和完整性。

还有一个关键字 **protected** 修饰的成员称为**保护成员**，外界也是无法访问的，只能被该类和从该类派生的派生类的方法访问。当然，类对象的所有成员都能被该类的友元(见 7.8 节)访问。

7.5 const 对象、const 成员函数、mutable 成员变量

7.5.1 const 对象和 const 成员函数

回顾 const 修饰的 const 对象和指针、引用的结合：

```
int main() {
    int i;
    const int ci = 1;
    ci = 3;                 //错: 不能修改 const 对象(变量)
    const int * p = &i;     //p 是指向 const int 即"常对象"的指针变量
```

```
    *p = 3;             //错：不能修改 p 指向的常对象(const 对象)
    const int &r = i;   //r 是 const int"常对象"的引用
    r = 3;              //错：不能修改 r 引用的常对象(const 对象)
}
```

即不能修改 const 对象、不能修改指针变量指向的或引用变量引用的 const 对象，即使初始化指针或引用变量的原来的变量是可以修改的。例如，不能修改 *p 的值。

如同内在类型，也可以定义类的 const 对象，但该 const 对象是不可以修改的(即不能修改该对象的数据成员)。同样，也不能修改指针变量指向的或引用变量引用的 const 类对象，即使初始化这些变量的原来对象是可以修改的。如：

```
class X {
public:
    int ival{ 0 };
};
int main() {
    const X x;          //x 是 const 对象,通过构造函数初始化
    std::cout << x.ival << std::endl;   //OK
    x.ival = 10;        //错：const 对象 x(的成员)不能被修改

    X y;
    y.ival = 10;        //y 不是 const 对象,当然可以修改它(的成员)
    const X *p = &y;    //p 指向 const X 对象
    p->ival = 20;       //错：不能通过 p 去修改它指向的 const X 对象
                        //即使初始化 p 的 y 是可以修改的
    const X &r = y;     //const X 的引用变量 r 绑定到 y
    r.ival = 20;        //错：不能通过引用变量 r 修改它引用的 const X 即"常对象"
                        //即使初始化 r 的 y 是可以修改的
}
```

因为 x 是 const 对象，所以不能被修改。因为 p 指向的是 const 对象，所以不能通过 *p 或->去修改 p 指向的那个对象。同理，也不能通过引用变量 r 修改 y(尽管 r 是用 y 初始化的)。

再看看 Date 类的 const 对象：

```
const Date day;
day.setYear(2008);          //错：不能去修改 const 对象 day
std::cout << day.getYear(); //错
```

通过 const 对象 day，不管是调用 setYear()还是 getYear()，都会产生编译错误。即使 getYear()函数确实不会修改 day，但编译器怎么知道它会不会修改 day 的数据呢？万一其中有代码修改了其数据成员呢？所以，编译器禁止这种函数调用。

难道通过 const 对象就不能调用类的成员函数如 getYear()吗？这不合理。

解决办法是修改 getYear()这种查询性的方法为 **const 成员函数**，即在函数的规范和函数体之间添加关键字 const。如下所示：

```
#include <iostream>
class Date {
    int year{ 2000 }, month{ 1 }, day{ 1 };
public:
    Date(int y = 2000, int m = 1, int d = 1):year(y), month(m), day(d) {
    }
    //委托 Date(int y = 2000, int m = 1, int d = 1)构造函数
    Date(int *p) :Date(p[0], p[1], p[2]) {}

    //get()函数: 访问类对象的数据
    int getYear()const { return year; }
    int getMonth()const { return month; }
    int getDay()const { return day; }

    //set 函数: 修改类对象的数据
    void setYear(int y) { if (y > 0) year = y; }
    void setMonth(int m) { if (m > 0) month = m; }
    void setDay(int d) { if (d > 0) day = d; }
    void print()const { std::cout << year << "-" << month << "-" << day << '\n'; }
};
int main() {
    const Date day;
    //day.setYear(2008);                    //错: 不能去修改 const 对象 day
    std::cout << day.getYear() << std::endl;  //Ok
}
```

其中不会修改数据成员的查询函数 getYear()、getMonth()、getDay()、print()都转换为了 **const 成员函数**,对 const 对象就可以调用这些 const 成员函数。将这些不会修改对象数据的函数设置为 const 成员函数也是一个良好的编程习惯。

假如将 setYear()这种修改性(修改数据成员)的函数定义为 const 成员函数(意图通过调用它们强行修改一个 const 对象)会怎么样呢?如将 setYear()修改为:

```
void setYear(int y) const  { if (y > 0) year = y; }
```

main()函数中如果调用这个函数:

```
day.setYear(2008);          //错: 不能去修改 const 对象 day
```

编译器仍然会报错:

```
error C3490:由于正在通过常量对象访问"year",因此无法对其进行修改
```

这说明编译器会自动检测这种不良行为,并禁止在 const 成员函数中修改数据成员,进一步说明了在 const 成员函数中是不能修改数据成员的。

7.5.2 重载 const

const 成员函数中的 const 关键字是函数签名的一部分,也就是说可以用于重载解析过

程中区分同名函数。因此,下列类 X 的 2 个函数 f()是完全不同签名的 2 个函数(即 f()是 2 个同名的重载成员函数),是可以同时作为一个类的成员函数的:

```
class X{
    void f() {/*...*/}
    void f()const {/*...*/}
}
```

下面的代码,试图通过 2 个 Date 对象调用 Date 类的 year()方法查询各自的年份:

```
#include <iostream>
class Date {
    int _year{ 2000 }, _month{ 1 }, _day{ 1 };
public:
    int& year() { return _year; }
    int& month() { return _month; }
    int& day() { return _day; }
};
int main() {
    Date day;
    day.year() = 2008;                          //ok
    std::cout << day.year() << std::endl;       //ok
    const Date day2;
    day2.year() = 2008;                 //编译错误.是合理的,因为不能修改 const 对象
    std::cout << day2.year() << std::endl;
                                        //编译错误.不合理,为什么不能查询 const 对象信息
}
```

编译程序,出现如下错误:

```
error C2662: "int &Date::year(void)": 不能将"this"指针从"const Date"转换为"Date &"
```

因为 day2 是 const 对象,而 year()函数不是 const 成员函数,不能通过 const 对象或 const 对象的指针或引用调用这个非 const 成员函数,只能通过非 const 对象或非 const 对象的指针或引用调用这个函数。

解决方法是重载 const,即定义一个重载的 const 函数。

```
#include <iostream>
class Date {
    int _year{ 2000 }, _month{ 1 }, _day{ 1 };
public:
    int& year() { return _year; }
    int& month() { return _month; }
    int& day() { return _day; }
    const int& year() const { return _year; }
    const int& month() const { return _month; }
    const int& day() const { return _day; }
};
```

```
int main() {
    Date day;
    day.year() = 2008;                      //没问题,调用的是非 const 函数: int& year()
    std::cout << day.year() << std::endl;   //ok
    const Date day2;
    //day2.year() = 2008;                   //编译错误.是合理的,因为不能修改 const 对象
    std::cout << day2.year() << std::endl;  //ok,没有编译错误
}
```

此时,day2.year()对 const 对象调用的 const 成员函数 year()就没有任何问题了。

注:当然可以用一种强制类型转换 **const_cast** 去掉一个对象的 const 性,然后去调用非 const 成员函数。但除特殊情形外,不建议这样用。

7.5.3 mutable 成员变量

虽然 const 成员函数不能修改类的一般数据成员,但用关键字 mutable 修饰的数据成员总是可以在 const 成员函数里被修改的。例如:

```
class X {
    mutable int count{ 0 };         //count 是 mutable 函数
    int ival{ 0 };
public:
    int val()const {
        count++;                    //mutable 成员总是可以被修改
        return ival;
    }
}
```

7.6 析构函数

和内在类型的变量一样,当一个类对象退出创建它的作用域时,该对象就被销毁,或者对于一个用 new 运算符动态创建的类对象,当使用 delete 运算符作用于它时,该对象也会被销毁。

当一个类对象被销毁时,会调用一个称为**析构函数(destructor)**的特殊成员函数。一个类只能有一个析构函数,如果类定义中没有定义这个析构函数,编译器会默认生成一个空函数体的析构函数,这个析构函数什么也不做。对于一个类类型 X,编译器生成的默认析构函数是:

```
~X(){}
```

一个类的**析构函数**的函数名是**类名**前加一个符号~。析构函数和构造函数一样,不能有返回类型,另外,析构函数也不能有任何形参。析构函数是用于销毁对象的,给它传递参数有什么意义呢?

对于前面的 Date 类，其析构函数是：

```
~Date(){}
```

当然也可以写成：

```
~Date() = default;
```

如果一个类的构造函数在创建对象时会申请某种资源(如打开一个文件或一个网络端口、申请一块内存)，那么就需要定义一个析构函数，当该类对象被销毁时，负责释放这个类对象占用的资源。

下面的 IntArray 类表示一个固定大小的数据元素是 int 的动态数组。

```
#include <iostream>
class IntArray {
    int * data{ nullptr };            //指针变量指向动态分配的内存块
    int size{0};                      //data 指向的动态数组的大小
public:
    IntArray(int s) :size(s) {
        data = new int[s];            //分配一块动态内存，地址保存在 data 中
        if (data)  size = s;
        std::cout << "构造了一个大小是" << s << "的 IntArray 对象\n";
    }
    ~IntArray() {
        std::cout << "析构函数\n";
        if (data) delete[] data;      //释放 data 指向的动态内存
    }
    void put(int i, int x) {
        if (i >= 0 && i < size) data[i] = x;
    }
    int get(int i) {
        if (i >= 0 && i < size) return data[i];
        else return 0;
    }
};
```

IntArray 类的构造函数申请一块形参 s 大小的 int 类型动态数组(data = new int[s];)，用于存储一些 int 类型的值，当该类对象被销毁时，会调用析构函数释放构造函数中申请的这块内存(delete[] data)。

下面代码定义了类 IntArray 的对象 arr，可存储 s 个 int 类型值，并通过 arr 的 get()和 put()去查询或修改下标是 i 的 int 类型值。

```
int main(){
    std::cout << "请输入人数: ";
    int s; std::cin >> s;
    IntArray arr(s);                  //创建 IntArray 对象，会为 s.data 成员分配一块动态内存
```

```
    std::cout << "请输入年龄: ";
    for (auto i = 0; i < s; i++) {
        int age;
        std::cin >> age;
        arr.put(i, age);
    }
    std::cout << "输入的年龄是: \n";
    for (auto i = 0; i < s; i++)
        std::cout << arr.get(i)<<'\t';
    std::cout << '\n';                    //这句结束后,将销毁 arr,因此 IntArray 的
                                          //析构函数释放 arr.data 指向的动态内存
}
```

下面是执行这个程序的运行情况。

```
请输入人数: 3
构造了一个大小是 3 的 IntArray 对象
请输入年龄: 21 23 28
输入的年龄是:
21      23      28
析构函数
```

析构函数能保证对象被销毁时,它申请的这块内存得到释放。如果一个对象被销毁时没有释放其占用的内存,会导致这块内存无法被其他程序或该程序的其他代码使用,即导致**内存泄漏**。如果程序中这些无法释放的动态内存块越来越多,会导致内存不足,使得操作系统的所有程序都无法正常运行。

7.7 静态成员

7.7.1 非静态成员变量和静态成员变量

前面见到的类的成员变量都是**非静态成员变量**。也就是说,类的每个对象都有一个自己单独的成员变量,这些成员变量是属于每个对象的。同样地,类的成员函数也都是一些**非静态成员函数**,这些函数必须通过具体的对象才能调用。

有时,需要和整个类而不是具体对象关联的成员。如可能定义属于整个类而不是单个对象的计数器变量,这个计数器变量是属于整个类或者说是类的所有对象共享的一个变量,每当创建一个类的对象时,这个计数器就自增 1,从而通过这个计数器就知道从这个类产生了多少对象。这种技术可以用于对一个共享资源的管理,如申请了一块内存或打开了某个文件,通过计数器就能知道这个共享资源被多少使用者使用,当计数器为 0 时就自动释放这个资源。如第 13 章将介绍的 C++ 的管理动态内存的智能指针就是用的这种技术。

为了定义这种整个类的所有对象共享的变量,需要用关键字 static 声明这个变量是一个所谓的**静态成员变量(static member variables)**。

例如：

```
class X{
private:
    static int count;              //count 是类 X 的静态成员变量,表示类 X 的对象的个数
public:
    int get_count() { return count;};
    X() { count++; }              //创建了一个新的 X 对象,count 增加 1
    ~X() { count--; }             //销毁了一个 X 对象,count 减少 1
};
```

上述代码中，X 类声明了一个叫作 count 的静态成员变量，当通过构造函数创建一个对象时，count++增加 1，当一个 X 对象销毁调用析构函数时，coun--减少 1。通过 X 类的这个 count 静态变量就可以知道程序中包含了多少 X 类型的对象。

但这个 count 不是属于单个的 X 对象而是整个 X 只有一个这样的 count 静态成员变量，因此，不能在类 X 的构造函数中对它初始化。类的静态成员变量和全局变量一样，整个程序只有一个，和全局变量不一样的是它属于类 X 的作用域。

类 X 定义中的静态成员变量 count 仅仅是声明还不是定义，因此，在类定义中不能对它初始化。如果将上述 count 变量的声明写成定义的形式：

```
static int count{0};
```

编译器会报告如下错误：

```
error C2864: X::count:带有类内初始化表达式的静态数据成员必须具有不可变的常量整型类型,或
必须被指定为"内联"
```

在哪里定义和初始化类的静态成员变量呢？必须在类体外定义并初始化它。

```
int X::count = 0;                 //或 int X::count{0};
```

注意：类体外定义静态成员变量其前面**不能有关键字 static**。

通常，类 X 的定义被放在某个头文件（如 X.h）中，这个头文件可能被多个需要使用 X 类的源文件（如扩展名是.cpp 的代码文件）包含。如果静态成员变量的 count 的定义语句也放在这个头文件中，就会导致这个静态成员变量被多次定义，违背了一次定义规则（ODR）。因此，这个静态变量的定义通常放在另外一个单独的.cpp 文件中，而不是和类 X 的定义放在同一个头文件中。

上述这种静态成员变量的声明和定义必须分开在不同的头文件和源文件中，显得很麻烦。

在 C++17 中通过关键字 **inline** 修饰一个静态变量，使得静态成员的声明与定义统一起来。例如：

```
static inline int count{0};       //X 对象的个数
```

即只要在类定义中一次性地定义（声明）这个静态成员变量而无须分开声明和定义，简化了

静态成员变量的定义。

尽管 count 是 private 的，在类体外定义它是可以的，但不能在类体外直接访问它，如：

```
X x;
int c = x.count;                    //错：count 是 private 的
c = X::count;                       //错：count 是 private 的
```

但可以通过类的成员函数如 get_count()去访问它：

```
int main() {
    X x,x2;
    X arr[3];
    std::cout << x.get_count()<< '\n';
}
```

上述代码通过 x 调用 get_count()返回静态成员变量 count 的值，程序结果：

```
5
```

通常将类的静态成员变量声明为 public(公开的)。如：

```
public:
static inline int count{};          //X 对象的个数
```

那么，就可以直接通过类对象或类去访问它：

```
int main() {
    X x,x2;
    X arr[3];
    std::cout << x.count << '\t' << arr[1].count << '\t' << X::count << '\n';
}
```

执行程序，输出结果：

```
5       5       5
```

7.7.2 静态常量

静态成员变量通常用于定义**常量**(**constant**)。在 C++17 之前不能直接在类定义中初始化一个静态常量，在 C++17 中，同样用 inline 关键字可以直接定义并初始化一个静态常量：

```
class Circle {
public:
    static inline const double PI{ 3.1415926 };
    Circle(double r) :radius(r) {}
    auto area() { return PI * radius * radius; }
private:
```

```
    double radius{ 0. };
};
```

上述代码中 Circle 定义了一个静态常量 PI。

7.7.3 静态成员函数

和静态成员变量类似，可以用 static 关键字说明一个类成员函数是一个**静态成员函数**而不是普通成员函数（非静态成员函数）。如：

```
class X{
public:
    static inline int count{};     //count 是类 X 的静态成员变量，表示类 X 的对象的个数
public:
    static int get_count() { return count; };     //静态成员函数
    X() { count++; }                //创建了一个新的 X 对象，count 增加 1
    ~X() { count--; }               //销毁了一个 X 对象，count 减少 1
};
```

和静态成员变量不同的是，静态成员函数的定义可以完全定义在类体内，当然也可以定义在类体外。

静态成员函数和静态成员变量都是属于整个类，通过类名就可以调用静态成员函数，而普通成员函数必须通过类对象才能调用。如：

```
int main() {
    X x, x2;
    X arr[3];
    std::cout << x.get_count() << '\t' << arr[1].get_count() << '\t'
          << X::get_count() << '\n';
}
```

静态成员函数 get_count()既可以通过类对象，也可以通过类作用域 X::去调用。程序的输出结果是：

```
5       5       5
```

7.7.4 类自身类型的静态成员变量

可以在一个类中定义类自身类型的静态成员变量。如对于 Date 类，可以定义一个表示默认日期的静态成员变量 default_date，所有 Date 对象如果没有提供初始化的年、月、日，就用这个默认的静态成员变量初始化。

```
#include <iostream>
class Date {
    int year{ default_date.year }, month{ default_date.month },
        day{ default_date.day };
```

```
public:
    const static Date default_date;
    Date(int y = default_date.year, int m = default_date.month,
        int d = default_date.day) :year(y), month(m), day(d) {
    }
    void print()const { std::cout << year << " - " << month << " - " << day << '\n'; }
};
```

上述代码在 Date 类中声明了一个 Date 类型的静态常量成员 default_date。还必须在类体外定义它：

```
const Date Date::default_date{2010,1,1};
```

注意：和普通静态成员变量不同，类自身类型的静态成员只能在类定义中声明且要在类定义外去定义它。不能用 inline 关键字统一声明和定义。

上述静态常量成员变量不能被修改。当然，也可以将它定义成非 const 类型的，这样就可以修改了，如下面的静态成员函数可以修改这个静态成员变量。

```
class Date {
    int year{ default_date.year }, month{ default_date.month },
        day{ default_date.day };
public:
    static Date default_date;
    static void set_default(const Date &d) {
        default_date.year = d.year;
        default_date.month = d.month;
        default_date.day = d.day;
    }
    Date(int y = default_date.year, int m = default_date.month,
        int d = default_date.day) :year(y), month(m), day(d) {
    }
    void print()const { std::cout << year << " - " << month << " - " << day << '\n'; }
};
```

然后在类体外，定义它：

```
Date Date::default_date{2010,1,1};
```

现在就可以修改这个静态成员变量了：

```
int main() {
    Date arr[3];
    Date d(2018,6,8);
    d.set_default(d);            //或 Date::set_default(d)
    Date arr2[5];
}
```

程序中 arr 数组的所有 Date 对象都初始化为 default_date{2010,1,1}的值,然后调用"d.set_default(d);"修改了这个 default_date,后面的 Date 对象如 arr2 的所有对象都默认初始化为这个新的 default_date 值。

7.8 友元

用 private 或 protected 声明的成员,除了类的成员函数外,外界是无法访问的,但如果在类中用关键字 **friend** 声明某个外部函数或外部类是这个类的**友元**,则这个外部函数或外部类是可以直接访问这个类的 private 成员。例如:

```
class Date{
    int _year{2000},_month{1},_day{1};
    friend void print(const Date&day); //外部函数 print()是 Date 类的友元
};
void print(const Date&day){              //print()可以访问 Date 对象 day 的私有成员
{     std::cout << day.year <<" - " << day.month <<" - " << day.day <<'\n';     }
}
void print2(const Date&day){             //print2()不能访问 Date 对象 day 的私有成员
{     std::cout << day.year <<" - " << day.month <<" - " << day.day <<'\n';     }
}
```

其中,外部函数 print()因为在类 Date 中被声明为 Date 类的友元,因此,可以直接访问类(对象)的私有成员(包括数据成员和函数成员),而 print2()函数则不行。因此,print2()函数内部的语句编译时会报错。

7.9 内联成员函数

和普通函数如果声明为内联(inline)函数可以提高程序效率一样,类的成员函数也可以是**内联(inline)成员函数**。如果一个类的成员函数是在类体内实现的,则这个函数就自动成为内联成员函数;如果一个类的成员函数是在体外实现的,则必须在类体内函数的声明前加关键字 inline,而类体外函数声明前不能加 inline。例如:

```
class X {
public:
    inline void f();
};
void X::f() {
    /* ... */
}
```

在类体内声明了 f()是内联成员函数,类体外函数声明前不能再有关键字 inline。

7.10 重新定义拷贝构造函数和赋值运算符函数

下面代码定义了一个表示字符串的类 String：

```
#include <iostream>
#include <cstring>                    //strlen()
class String {
    char * data{ nullptr };
    int size_{0};
public:
    String() = default;
    String(const char * s) {
        auto len = strlen(s);
        data = new char[len + 1];     //分配一块存储字符的动态内存块
        if (!data) return;
        strcpy(data, s);              //拷贝字符串内容从 s 到 data 指向的空间
        size_ = len;
    }
    auto size() { return size_; }
};
```

然后，可能这样使用它：

```
int main() {
    String s("hello world");
    String s2(s);                     //调用拷贝构造函数,将 s 拷贝到新对象 s2 中
}
```

运行程序时出现非法内存访问导致的程序崩溃。

这是由于 s2 是 s 的硬拷贝，因此，它们的 data 将指向同一块内存，那么 s2 先销毁时会调用析构函数释放这块内存，然后 s 开始销毁，又要释放同一块内存，而释放一个已经释放的内存的后果是灾难性的。

同样，下面的代码也因为同样的原因而导致程序崩溃。因为 s3 和 s 也是指向了同一个内存，它们销毁时都会释放这块内存，导致同一块内存被多次释放，程序一样会崩溃。

```
int main() {
    String s("hello world");
    String s3;
    s3 = s;
}
```

解决的办法是重新定义拷贝构造函数和赋值运算符。

```
class String{
    //...
    String(const String&s);
```

```
    String& operator = (const String&s);
};
String::String(const String& s){
    std::cout <<"拷贝构造函数:\n";     //仅仅用于输出是否进入该函数
    if(s.data == 0)return;
    if(this!= &s){
      data = new char[s.size + 1];    //申请一块自己专用的内存块
      if(!data) return;
      size = s.size;
      strcpy(data,s.data);            //拷贝字符串内容
    }
}

String& String::operator = (const String& s){
    if(this!= &s){
      std::cout <<"赋值运算符:\n";    //仅仅用于输出是否进入该函数
      if(s.data == 0)return;
      char *p = new char[s.size + 1]; //申请一块自己专用的内存块
      if(!p) return;
      delete[ ]data;                  //释放原来的内存块
      data = p;                       //data 指向新的内存块
      size = s.size;
      strcpy(data,s.data);            //拷贝字符串内容
    }
    return * this;
}
```

拷贝构造函数和赋值运算符函数都重新分配了一块单独的内存块存放复制的数据，因此，每个对象的 data 都指向自己单独的内存块，这些对象被销毁时，程序不会因为释放同一块内存而崩溃。

7.11 实战：线性表及应用

7.11.1 线性表

类对象和基本数据类型的变量在 C++中都称为对象，用来表示一个具有完整的独立含义的**数据元素**。一个程序的数据不仅仅是各种类型的单个的零星的数据元素，还经常会出现一组同类型的数据元素，如表 7-1 所示。

表 7-1 学生信息表

学号	姓名	性别	籍贯	年龄
98131	张三	男	北京	20
98164	李斯	女	上海	21
98165	王武	男	广州	19
98182	赵柳	女	香港	22
98224	…	…	…	…

显然不可能给每个数据元素定义一个单独名字的变量,这就需要将这些数据以某种方式有条理地存储在计算机内存中,对这组数据还可能进行插入、删除、查找、排序等处理操作。如何高效率地存储、表示和处理一组数据元素决定了程序的效率和性能,也是计算机专业最重要的课程“数据结构”的研究内容。

在数据结构中,“线性表”是一种最基本、最常用的数据结构,是日常工作生活中广泛使用的各种表单(如学生名单、通讯录、工资单、商品列表、聊天记录、搜索引擎的搜索结果列表)的逻辑抽象。它刻画的是数据元素之间的一种线性关系,即这些数据元素可以排成一列,所有数据元素构成一个**线性序列**。每个元素都对应一个确定的序号,例如第 1 个元素的序号是 1、第 2 个元素的序号是 2、……。因此,线性表的定义是“有限个元素的序列”。可以表示成如下抽象形式:

$$(a_1, a_2, \cdots, a_n)$$

线性表的特点:数据元素具有一个挨一个的关系。即每个数据元素最多只有一个“直接前驱”,最多只有一个“直接后继”。如 a_2 的“直接前驱”是 a_1,a_2 的“直接后继”是 a_3,第 1 个元素没有直接前驱,最后 1 个元素没有直接后继。

在研究“线性表”这种数据结构时,不关心具体数据元素是什么(即不关心具体是什么表),而是研究这种线性表具有的抽象特性。从面向对象角度看,$(a_1, a_2, \cdots, a_n)$就是线性表的数据属性。此外,还有一些对线性表的操作属性(即功能属性),例如下面是一些常见的对线性表的操作。

初始化:创建一个空的线性表()。

插入元素:在线性表的某个位置插入一个新的数据元素,如(a)→(a, b)→(a, c, b)。

删除元素:删除线性表的某个元素。如(a,c,b)→(a, b)。

读写元素:读取或修改某个元素的值。对(a,c,b),操作 get(2)返回 c;操作 set(2, e)将第 2 个元素修改成 e,即(a,e,b)。

查找元素:查询是否存在满足某条件(如相等)的元素。对(a,c,b),操作 find(c)返回 2。

查询表长:查询线性表中的数据元素的数目。对(a,c,b),操作 size()返回 3。

上述是从抽象逻辑角度描述了线性表是一个什么样的数据结构,即从数据属性上看,它是一个线性序列(结构);从操作上看,它具有初始化、插入、删除、读写等操作。这种数据结构的抽象描述称为**逻辑结构**。逻辑结构和计算机实现无关,为了将逻辑结构在程序中表示出来,还要研究**物理结构**(也称为**存储结构**),即如何将数据元素及其逻辑关系在计算机内存中表示出来并且用程序算法实现这些逻辑操作。

实际上,C++语言自带的数组已经提供了存储表示线性表的功能。如:

```
char a[ ]{'a', 'b', 'c'};
```

并且可以根据下标读写某个数据元素。如:

```
a[1] = 'e';
```

但并没有实现线性表的其他常用操作,例如,无法动态插入、删除。C++内在数组必须

一开始就固定数组的大小,而实际问题的线性表的大小是无法预期的且是可以动态变化的。

C++标准库提供了许多数据结构,包括线性表的实现,如 std::vector 和 std::list 分别提供了以动态数组方式和链表方式的 2 种线性表的实现。数组方式实现的线性表称为**顺序表**,而链表方式实现的线性表称为**链表**。

下面通过实现顺序表和链表来揭示 std::vector 和 std::list 的实现原理。

7.11.2 线性表的顺序实现:顺序表

顺序表就是用一块连续的存储空间(即数组)来存储线性表的元素,以物理位置的相邻性来表示逻辑上的相邻关系,并在此基础上实现线性表的基本操作。即逻辑上相邻的元素在物理存储地址上也是紧邻的。如线性表(a_1, a_2,…, a_n)的物理存储如图 7-3 所示。

这块连续存储空间可以用 C++的动态内存分配申请。假如数据元素类型是 ElemType,可以开始分配一定容量(如 capacity=10)的存储空间并保存在某个变量(如 data)中:

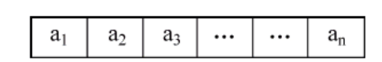


图 7-3 线性表(a_1,a_2,…,a_n)的物理存储

```
ElemType * data = new ElemType[capacity];
```

这块动态数组空间是用来存储数据元素的,一开始时还没有数据元素,随着今后的插入或删除操作,数据元素的数目会不断改变,所以还需要一个变量 n 表示实际元素的个数。

因此,表示一个线性表需要 3 个变量:空间的起始地址(data)、空间的容量(capacity)、实际数据元素的个数(n)。除这些数据变量外,对线性表还有插入、删除等操作,因此,可以用一个类来表示数据变量(成员变量)和操作(成员函数):

```
using ElemType = char;                //假设数据元素类型 ElemType 是 char 类型
class Vector {
    ElemType * data{ nullptr };       //空间起始地址
    int  capacity{ 0 }, n{ 0 };       //空间容量和实际元素个数,初始化为 0
public:
    Vector(const int cap = 5)         //创建容量是 cap 的一个线性表
        :capacity{ cap }, data{ new ElemType[cap] }    {}
    bool insert(const int i, const ElemType &e);       //在 i 处插入元素
    bool erase(const int i);                           //删除 i 元素
    bool push_back(const ElemType &e);                 //在表的最后添加一个元素
    bool pop_back();                                   //删除表的最后一个元素

    bool get(const int i, ElemType &e)const;           //读取 i 元素
    bool set(const int i, const ElemType e);           //修改 i 元素
    int size() const { return n; }                     //查询表长
private:
    bool add_capacity();                               //扩充容量
};
```

上述代码首先假设数据类型 ElemType 是 char 类型:

```
using ElemType = char;
```

其中，构造函数完成初始化一个线性表的数据成员的工作，即申请数据元素 ElemType 类型的大小是 cap 的动态数组内存，即这块空间可存储 cap 个 ElemType 类型的元素，因此容量成员是 capacity{cap}，但开始实际数据元素个数是 n{0}，即还没有任何数据元素。图 7-4 所示是一个 Vector 类对象的内存布局。

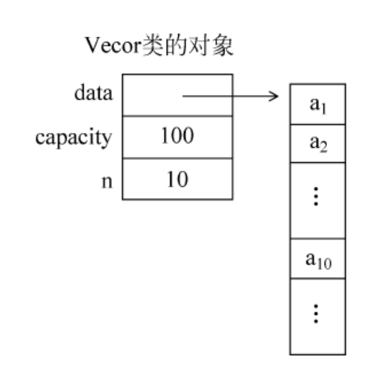


图 7-4　Vector 类对象的内存布局

查询表长的 size()函数很简单，直接返回记录的表长变量 n。

随着不断插入数据元素，可能分配空间已经满了，需要增加容量，因此还有一个私有函数 add_capacity()专门用于增加容量，即分配更大的内存空间。先来看看这个函数的实现：

```
bool Vector::add_capacity() {
    ElemType * temp = new ElemType[2 * capacity];    //分配 2 倍大小的更大空间
    if (!temp) return false;                         //申请内存失败
    for (auto i = 0; i < n; i++) {                   //将原空间 data 数据复制到新空间 temp
        temp[i] = data[i];
    }
    delete[] data;                                   //释放原来空间内存
    data = temp;                                     //data 指向新的空间 temp
    capacity *= 2;                                   //修改容量
    return true;
}
```

该函数先分配更大空间(temp = new ElemType[2 * capacity])，然后将 data 指向的原空间数据复制到 temp 指向的新空间的对应元素中，并释放原来空间(delete[] data)，让 data 指向新空间(data= temp)，最后修改表示容量的变量(capacity *= 2)。该函数返回 true 或 false 表示成功或失败。

下面再看如何在一个位置 i 插入一个新元素(注意这里的 i 是从 0 开始的下标而不是序号)：

```
bool Vector::insert(const int i, const ElemType &e) {
    if (i < 0 || i >= n) return false;               //插入位置合法吗
    if (n == capacity && !add_capacity())            //已满，增加容量
        return false;
    for (auto j = n; j > i; j--)                     //将 n-1 到 i 的元素都向后移动一个位置
        data[j] = data[j - 1];                       //j-1 移到 j 位置上
    data[i] = e;
    n++;                                             //不要忘记修改表长
    return true;
}
```

该函数首先判断插入位置是否合法，然后如果空间已满就调用 add_capacity()增加容量(if (n == capacity && ! add_capacity()))。接着，将下标从 n−1 到 i 的元素都向后移动一个位置，这样，i 位置相当于空开了，就可以将新元素放入到该位置(data[i] = e)。最后不要忘记数据元素个数增加了 1(n++;)。插入过程如图 7-5 所示：

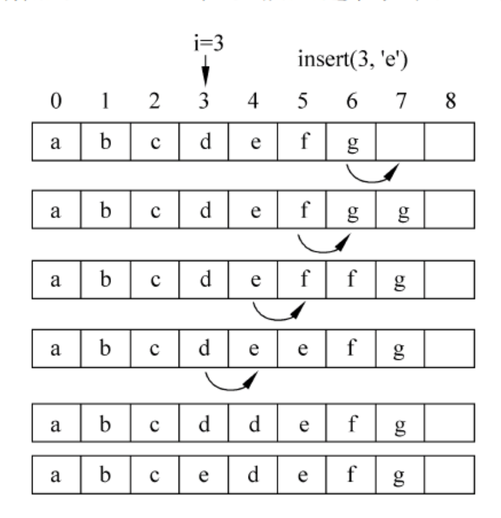


图 7-5 在下标 i=3 的位置插入一个元素'e'

注意：n 个元素的下标是 0、1、……、n−1。

删除下标是 i 的元素过程与其类似，就是将下标 i+1 到 n−1 的每个元素依次向前移动一个位置，代码如下：

```
bool Vector::erase(const int i) {
    if (i < 0 || i >= n) return false;          //位置 i 合法吗
    //i+1 到 n-1 元素依次向前移动一个位置
    for (auto j = i; j < n - 1; j++)
        data[j] = data[j + 1];                  //j+1 移到 j 位置上
    n--;                                        //不要忘了: 表长变量减去 1
    return true;
}
```

在表尾插入或删除元素不需要移动元素，因此速度非常快，为此提供单独的 2 个函数 push_back()(用于表尾插入一个元素)和 pop_back()(用于删除表尾元素)：

```
bool Vector::push_back(const ElemType &e) {
    if (n == capacity && !add_capacity())      //空间满就扩容
        return false;
    data[n++] = e;                              //e 放入下标 n 位置,然后 n++
    return true;
}
bool Vector::pop_back() {
    if (n == 0) return false;                   //空表
    n--;                                        //n 减去 1 就相当于删除了表尾元素
    return true;
}
```

根据下标读写元素的函数很简单：

```
//读取 i 元素的值,放入引用变量 e 中
bool Vector::get(const int i, ElemType &e)const{
    if (i >= 0 && i < n) {                      //下标是否合法
        e = data[i]; return true;
    }
    return false;
}
//修改 i 元素的值
bool Vector::set(const int i, const ElemType e){
    if (i >= 0 && i < n) {                      //下标是否合法
        data[i] = e; return true;
    }
    return false;
}
```

现在，可以编写一个测试程序，测试上述线性表的实现(操作)是否正确：

```
#include <iostream>
void print(const Vector &v) {                   //输出线性表中的所有元素
    ElemType e;
    //遍历每一个下标 i:0,1,…,size()-1
    for (auto i = 0; i != v.size(); i++) {
        v.get(i, e);                            //通过 e 返回下标 i 处的元素值
        std::cout << e << '\t';
    }
    std::cout << std::endl;
}
int main() {
    Vector v(2);                                //创建容量是 2 的空线性表
    v.push_back('a');                           //线性表最后添加一个元素'a'
    v.push_back('b');                           //线性表最后添加一个元素'b'
    v.push_back('c');
    v.insert(1, 'd');                           //下标 1 处插入一个元素'd'
    print(v);
    ElemType e;
    v.get(1, e);                                //查询下标 1 处的元素值
    std::cout << e << '\n';
    v.set(1, 'f');                              //修改下标 1 处的元素值为'f'
    print(v);
    v.erase(2);                                 //删除下标 2 处的元素
    print(v);
    v.pop_back();                               //删除最后一个元素
    print(v);
}
```

执行程序，输出结果：

```
a    d    b    c
d
a    f    b    c
a    f    c
a    f
```

作为练习，读者可以为线性表添加一个查找功能的 find()成员函数。

7.11.3　线性表的链式实现：链表

在线性表中插入、删除数据元素时需要移动很多元素，如果数据元素本身很大，就比较耗时。另外，如果线性表的元素个数非常多，可能无法申请到一个足够大的内存块，为了克服这些缺点，可以考虑采用所谓的**链表**来表示线性表。

如图 7-6 所示，每个数据元素用一个单独的内存块来表示，这个内存块称为**结点**，每个结点除存储这个元素的值外，还有一个指针变量指向逻辑上的下一个元素的结点的位置，即指针变量存储下一个元素结点的地址。

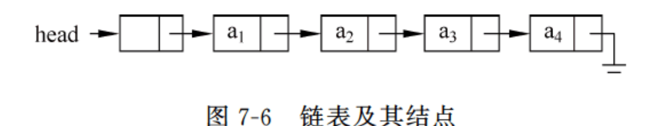


图 7-6　链表及其结点

因为每个元素的结点内存块是用 new 单独申请的，因此，逻辑上相邻的数据元素的结点的物理位置通常是不相邻的，甚至可能在内存空间中相距很远。数据元素之间的逻辑关系（相邻关系）是通过结点内的指针表示的。

每个结点本身是没有名字的，但因为有指针将它们串在一起，因此，只要将第一个元素结点的地址保存到一个变量（如图 7-6 中的 head 变量）中就可以了。

由于第一个元素没有前驱元素，如果直接将第一个元素结点保存在一个变量中，今后执行线性表的操作时，需要区分操作的结点是否是第一个元素结点还是其他元素结点，这给操作的实现带来不便。因此，一般地，会在第一个元素结点前再添加一个不含任何数据的辅助结点，称为**头结点**，而第一个元素的结点则称为**首结点**，即首结点前面是一个头结点，然后将头结点的地址保存在一个指针变量（如图 7-6 中的 head）中，通过这个指向头结点的指针变量就能顺藤摸瓜找到所有的结点。最后一个元素的结点称为**尾结点**。

首先定义表示一个数据元素的结点类型，其中保存的是数据元素的值和下一个结点的地址。即定义如下叫作 LNode 的类：

```
using ElemType = char;                    //假设数据元素的类型是 ElemType
struct LNode {                            //struct 定义的类的成员默认是公开的
    ElemType data;
    LNode * next{nullptr};                //next 是指向下一个元素结点的指针变量
};
```

然后可以定义表示整个链表的类 List，可将结点类 LNode 作为其内部嵌套类（即可以在一个类的内部定义另外的类，称为**嵌套类**）。

```
using ElemType = char;                       //假设数据元素的类型是 ElemType
class List {
    struct LNode {                           //struct 定义的类的成员默认是公开的
        ElemType data;
        LNode * next{nullptr};               //next 是指向下一个元素结点的指针变量
    };
    LNode * head;                            //指向头结点的指针变量
public:
    //初始化一个不含任何数据只有头结点的空的链表
    List() :head{ new LNode{} } {}

    bool insert(const int i, const ElemType &e);   //在 i 处插入元素
    bool erase(const int i);                 //删除 i 元素

    bool push_back(const ElemType &e);       //表的最后添加一个元素
    bool pop_back();                         //删除表的最后元素

    bool push_front(const ElemType &e);      //插入元素成为第一个元素(首结点)
    bool pop_front();                        //删除首结点

    bool get(const int i, ElemType &e)const; //读取 i 元素
    bool set(const int i, const ElemType e); //修改 i 元素
    int size() const;                        //查询表长
};
```

List 类中只有一个数据成员即存储链表头结点的指针变量 head。在构造函数中创建一个只含头结点的空表。该函数只需将新申请的头结点地址保存在 head 变量(head{new LNode{}})中,如图 7-7 所示。

另外和 Vector 类比较,多了 2 个成员函数 push_front()和 pop_front(),这 2 个函数可以用于快速在链表头部插入和删除一个结点。而 push_back()和 pop_back()则需要每次从链表头走到链表尾才能操作。

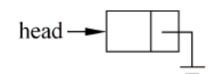


图 7-7 空的链表

为了向空的链表中添加元素,可以先实现最简单的 push_front():

```
bool List::push_front(const ElemType &e) {
    LNode *p = new LNode;    //创建一个新的结点
    if (!p) return false;    //分配内存失败
    p->data = e;             //将新数据放入 p 指向的新结点的 data 成员中
    p->next = head->next;//p 指向结点的 next 指针指向原来的首结点
    head->next = p;          //head 指向结点的 next 指向 p 指向的新结点,即新结点成为首结点
}
```

首先动态申请一个结点内存(LNode * p = new LNode;),如成功,指向结点就将新数据值 e 放入 p 指向的新的结点的 data 数据成员中(p->data = e;)。然后使 p 的 next 指针指向原来的首结点(p->next = head->next;),也就是说原来的首结点(head->next 指

向的结点)挂在 p 指向结点的后面了。最后修改 head 指向结点的 next 变量等于 p(head—>next = p;),即都指向新结点,从而新结点成为挂在头结点之后的首结点。执行过程如图 7-8 所示。

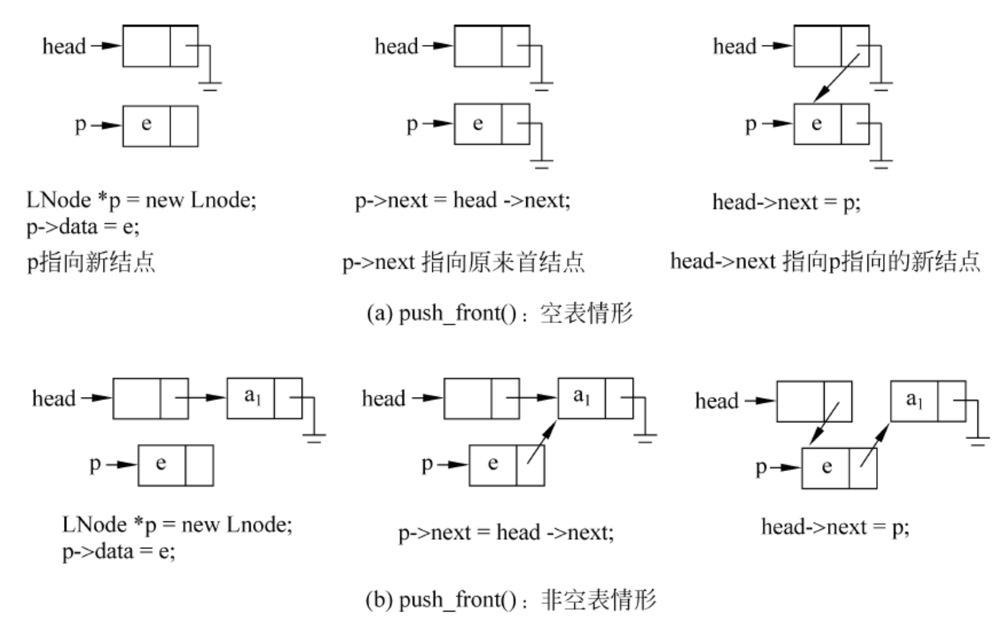


图 7-8　链表的前插：push_front()

删除首结点 pop_front()的过程是：如果是空表,则直接返回(if (! head—> next) return false;),否则先将要删除的首结点保存在临时的指针变量 p 中(LNode *p = head—>next;),然后修改 head 指向结点的 next 指向 p 指向结点的后一个结点(head—> next = p—>next;),即跳过首结点；最后销毁 p 指向的原来的首结点(delete p;),如图 7-9 所示。

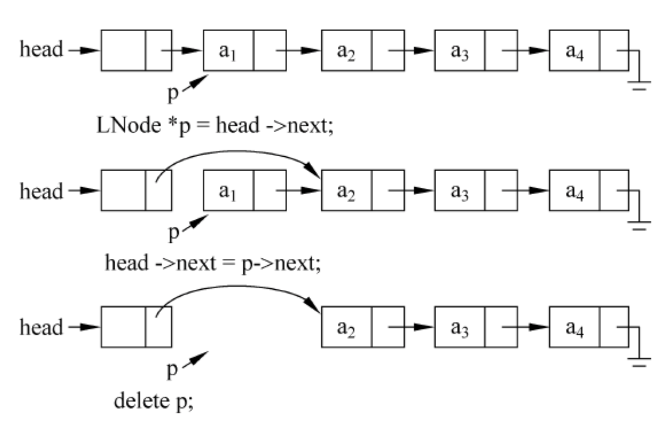


图 7-9　删除链表的首结点：pop_front()

pop_front()代码如下：

```
bool List::pop_front() {
    if (!head->next) return false;      //空表
    LNode *p = head->next;              //p 指向要删除的首结点
    head->next = p->next;               //head 的 next 指向 p 的后一个结点,即跳过首结点
    delete p;                           //删除原来的首结点
```

```
        return true;
    }
```

如何得到链表中的数据元素个数(即链表的表长)? 很简单,就是从头结点走到尾结点,看看遇到了多少数据结点就可以了(如图 7-10 所示):

```
int List::size()const {
    LNode *p{head}; auto i{ 0 };          //p 指向头结点,计数器 i 为 0
    p = p->next;                          //p 指向首结点
    while (p) {                           //p 不是空指针,表示遇到一个结点
        i++;                              //计数器增加 1
        p = p->next;                      //p 指向下一个结点
    }
    return i;
}
```

即用一个指针变量 p 先指向头结点(LNode *p{head}),计数器 i 是 0。然后将 p 移向下一个结点(p = p->next;),只要 p 不是空指针,就增加计数器(i++;),并将 p 移向下一个结点(p = p->next;)。重复这个循环过程,直到 p 为空指针,这个时候 i 的值就是走过的结点个数(不包括头结点)。

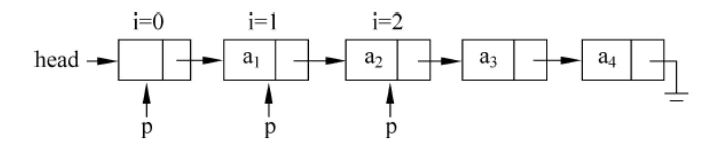


图 7-10　求链表的表长: size()

和顺序表可以根据下标(或序号)直接定位到相应数据元素的位置不同,链表必须从 head 头结点指针开始一个一个地移动,并用一个计数器记录走过的数据元素,才能定位到某个序号(如 i)对应的数据元素结点。根据序号 i 读、写、插入、删除元素都需要这个根据序号定位的功能,因此,可以给 List 类添加一个私有的辅助成员函数 **locate(const int i)** 用于定位序号是 i 的数据元素。其过程(如图 7-11 所示)类似于 size()函数。代码如下:

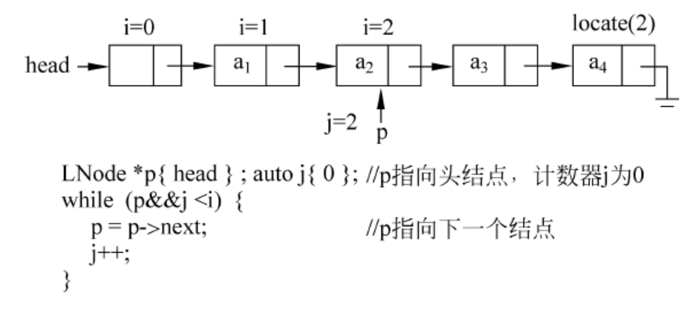


图 7-11　链表的定位: locate(const int i)

```
List::LNode * List::locate(const int i)const{
    if (i < 0) return nullptr;            //插入位置不合法
    LNode *p{ head }; auto j{ 0 };        //p 指向头结点,计数器 i 为 0
```

```
        while (p&&j < i) {
            p = p->next;                //p 指向下一个结点
            j++;
        }
        return p;
    }
```

即只要指针 p 不是空指针且计数器还没有到达 i(while (p&&j < i)),就一直将指针后移(p = p->next;)并增加计数器(j++;)。最后返回的指针 p 如果是空指针,则说明没找到;如果非空,则 p 就指向序号为 i 的结点。

如何删除(erase)序号 i 的结点?如图 7-12 所示,其方法使先定位置到序号为 i-1 的结点(LNode *p = locate(i-1)),然后如同 pop_front()那样删除第 i 个结点。即先用临时指针变量保存要删除的结点 i 的地址(LNode *q = p->next;),然后将 i-1 号结点的 next 指针进行修改,跳过这个 i 号结点(p->next = q->next;),最后销毁这个 i 号结点(delete q;)。

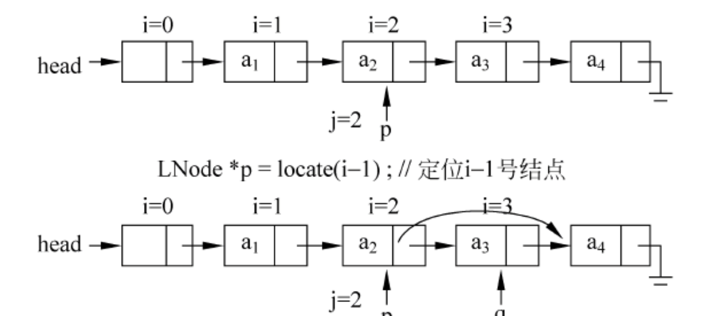


图 7-12 链表的删除:erase(3)

代码如下:

```
bool List::erase(const int i) {          //删除 i 元素
    LNode *p = locate(i-1);              //定位 i-1 号结点
    if (p) {                             //如果 p 指向的第 i-1 号结点存在
        LNode *q = p->next;              //q 保存要删除的结点地址
        p->next = q->next;               //使 p 指向结点的 next 指针跳过 q 指向的结点
        delete q;                        //删除 q 指向的那个结点
        return true;
    }
    return false;                        //i 超出了表长
}
```

在序号 i 位置插入一个元素(insert)的过程是:先定位到第 i-1 号位置(LNode *p = locate(i - 1);),为新元素申请新结点(LNode *q = new LNode;),然后采用类似于 push_front()的插入过程修改相应结点的指针即可,如图 7-13 所示。

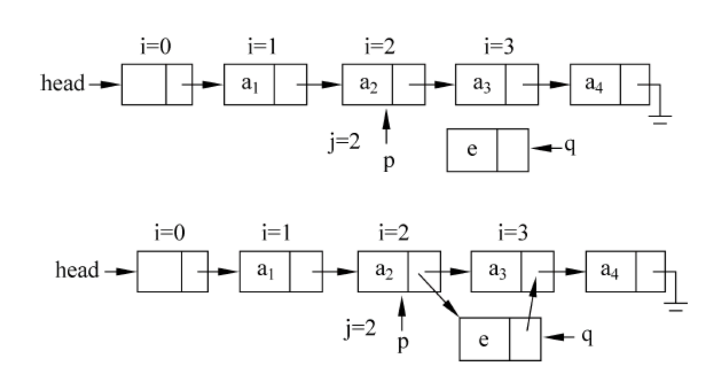


图 7-13 链表的插入：insert(3)

代码如下：

```
bool List::insert(const int i, const ElemType &e) {
    LNode *p = locate(i - 1);          //定位 i-1 号结点
    if (p) {                           //p 指向的第 i-1 号结点存在
        LNode *q = new LNode;          //q 指向分配的新结点内存块
        if (!q) return false;
        q->data = e;
        q->next = p->next;             //将 p 指向结点的后继结点挂到 q 指向结点的后面
        p->next = q;                   //将 q 指向的结点挂到 p 指向结点的后面
        return true;
    }
    return false;
}
```

根据序号的读写操作 get()和 set()也是先定位到序号 i 的元素结点(LNode *p = locate(i);),然后读写该结点的 data(p->data)：

```
bool List::get(const int i, ElemType &e)const {
    LNode *p = locate(i);              //定位 i 号结点
    if (p) {
        e = p->data; return true;
    }
    return false;
}
bool List::set(const int i, const ElemType e) {
    LNode *p = locate(i);              //定位 i 号结点
    if (p) {
        p->data = e; return true;
    }
    return false;
}
```

读者可以模仿上述过程实现剩余的 2 个函数 push_back()和 pop_back(),即在链表的尾部添加和删除一个元素。

编写好这个 List 类,还应测试一下它的实现是否正确,可以将前面对 Vector 的测试代

码稍做修改：

```
#include <iostream>
void print(const List &v) {
    ElemType e;
    for (auto i = 1; i <= v.size(); i++) {  //这里的 i 是序号不是顺序表的下标
        v.get(i, e);
        std::cout << e << '\t';
    }
    std::cout << std::endl;
}

int main() {
    List v;
    v.push_front('a');
    v.push_front('b');
    v.push_front('c');
    print(v);
    v.insert(1, 'd');
    print(v);
    ElemType e;
    v.get(1, e);
    std::cout << e << '\n';
    v.set(1, 'f');
    print(v);
    v.erase(2);
    print(v);
    v.pop_front();
    print(v);
}
```

执行程序，结果如下：

```
c       b       a
d       c       b       a
d
f       c       b       a
f       b       a
b       a
```

其中的 print()函数遍历整个链表时，对于每个序号 i get(i,e)都要从头开始走到这个 i 结点位置，导致很多重复的工作。如何改进它？可以给 List 再添加一个 LNode * 类型的数据成员 current 指向当前正访问的结点，另外添加 2 个成员函数用于将 current 定位到第一个元素和向后移动 current 指针。

bool First(ElemType &e)：用于定位第一个元素，即 current 指针指向第一个元素。

bool Next(ElemType &e)：用于返回当前元素，并将内部的 current 指针向后移动一个位置。

感兴趣的读者可以尝试实现这样的函数，并用它去遍历整个链表，提高遍历算法的效率。

7.11.4 实现一个图书管理的程序

假设要编写一个图书管理程序，其中的数据元素就是图书，所有的图书可以用一个线性表来存储管理。

假如采用的是顺序表（链表也是一样的），只要将线性表的 ElemType 换成图书类型，Vector（或 List）的代码不需要做任何修改：

```
#include <string>
class Book {
public:
    std::string name, author, publisher;
    double price;
};
using ElemType = Book;              //ElemType 定义为 Book 类型
//using ElemType = char;
class Vector {
    //…
};
```

借助于 Vector，可以很容易地写出一个简单的图书管理程序，示范程序如下：

```
#include <iostream>
//输入一个数据元素的辅助函数
void input(ElemType &e) {
    std::cout << "请输入图书的信息：书名 作者 出版社 价格:\n";
    std::cin >> e.name >> e.author >> e.publisher >> e.price;
}
//打印一本图书信息
void print(const ElemType &e) {
    std::cout << e.name << e.author << e.publisher << e.price << '\n';
}
//打印 Vector 对象 v 中的所有图书信息
void print(const Vector &v) {
    ElemType e;
    for (auto i = 0; i != v.size(); i++) {
        v.get(i, e);
        print(e);                   //输出该图书信息
    }
    std::cout << std::endl;
}
//帮助提示函数
void help() {
    std::cout << "请输入命令:i(插入)、e(删除)、a(追加)、b(删除最后元素)、\n";
```

```
    std::cout << "s(删除某序号元素)、g(查询某序号元素)、p(打印)\n";
}
int main() {
    Vector books;
    ElemType e;
    char cmd;
    help();
    while (std::cin >> cmd) {
        if (cmd == 27)break;
        else if (cmd == 'I'|| cmd == 'i') {          //插入一本图书
            std::cout << "请输入插入的位置(从 0 开始): ";
            int i; std::cin >> i;
            input(e);
            books.insert(i, e);
        }
        else if (cmd == 'e'|| cmd == 'E') {          //删除一本图书
            std::cout << "请输入删除的位置(从 0 开始): ";
            int i; std::cin >> i;
            books.erase(i);
        }
        else if (cmd == 'a'|| cmd == 'A') {          //在最后插入一本图书
            input(e);
            books.push_back(e);
        }
        else if (cmd == 'b'|| cmd == 'B') {          //删除最后一本图书
            input(e);
            books.pop_back();
        }
        else if (cmd == 's'|| cmd == 'S') {          //修改某序号的图书
            std::cout << "请输入要修改的图书的位置(从 0 开始): ";
            int i; std::cin >> i;
            input(e);
            books.set(i, e);
        }
        else if (cmd == 'g'|| cmd == 'G') {          //查询某序号的图书
            std::cout << "请输入要查询的图书的位置(从 0 开始): ";
            int i; std::cin >> i;
            books.get(i, e);
            print(e);
        }
        else if (cmd == 'p'|| cmd == 'P') {          //显示所有图书
            print(books);
        }
        help();

    }
}
```

执行程序,输出结果:

```
请输入命令:i(插入)、e(删除)、a(追加)、b(删除最后元素)、
s(修删除某序号元素)、g(查询某序号元素)、p(打印)
a
请输入图书的信息: 书名 作者 出版社 价格:
C语言 dong 清华 45
请输入命令:i(插入)、e(删除)、a(追加)、b(删除最后元素)、
s(修删除某序号元素)、g(查询某序号元素)、p(打印)
a
请输入图书的信息: 书名 作者 出版社 价格:
数据结构  严 清华  39
请输入命令:i(插入)、e(删除)、a(追加)、b(删除最后元素)、
s(修删除某序号元素)、g(查询某序号元素)、p(打印)
i
请输入插入的位置(从 0 开始): 1
请输入图书的信息: 书名 作者 出版社 价格:
Python 李 电子工业  39
请输入命令:i(插入)、e(删除)、a(追加)、b(删除最后元素)、
s(修删除某序号元素)、g(查询某序号元素)、p(打印)
p
C语言 dong 清华 45
Python 李电子工业 39
数据结构严清华 39

请输入命令:i(插入)、e(删除)、a(追加)、b(删除最后元素)、
s(修删除某序号元素)、g(查询某序号元素)、p(打印)
```

7.12 实战:面向对象游戏——基于链表的贪吃蛇游戏

贪吃蛇游戏是一款经典的益智游戏,该游戏通过控制蛇头方向吃蛋,从而使得蛇变得越来越长。

游戏的玩法规则:用户控制蛇的上、下、左、右前进方向,寻找吃的东西,每吃一口就能得到一定的积分,蛇的身子会变长,蛇在运动过程中不能碰墙,不能咬到自己的身体,身子越长玩的难度越大,等到了一定的分数,就能过关,然后继续玩下一关。

7.12.1 面向对象游戏引擎

一个游戏初始化后,首先出现的是一个主界面,然后游戏中的精灵们相互作用,也接受用户的输入,使得游戏环境发生变化并以绘制的场景图像显示出来。每个游戏都具有一些共同数据属性:游戏画面窗口、背景中有一些精灵等对象。每个游戏都包含下面的一些共同工作:初始化、事件处理、更新数据、绘制场景。可以将所有游戏都共有的这些数据属性和功能属性用一个类表示,这个类可称为**游戏引擎**类,因为它控制着整个游戏的运行过程。

这个游戏引擎类中包含窗口、精灵等对象,每个对象也都具有自己的数据属性和功能属

性，如窗口有长宽、标题、背景和前景颜色，可以在窗口的画布上绘制像素等，精灵有自己的位置、图像、速度、运动等属性。游戏引擎和窗口、精灵之间是一种包含关系。

游戏引擎类主要包含的属性如下。

1. 数据属性

(1) 游戏窗口(屏幕)Window：窗口长宽、标题、绘制表面、背景或前景图像或颜色、字体颜色等。

(2) 背景。

(3) 所有的精灵 Spirite。

2. 功能属性

(1) 初始化。

(2) 游戏主循环 run()。

- 事件处理 processEvent()。
- 更新数据 update()、碰撞检测处理 collision()。
- 绘制场景 render()。

(3) 退出游戏 quit()。

可以定义如下的 GameEngine 游戏引擎类，其中构造函数完成游戏引擎的初始化工作，包括游戏窗口图形环境的初始化和游戏数据的初始化。然后是游戏的主循环 run()方法，其中不断重复 4 个过程：事件处理 processEvent()、更新数据 update()、碰撞检测处理 collision()、绘制场景 render()。除了这些主要方法外，GameEngine 还可以定义其他一些辅助方法，如退出游戏的清理工作的 quit()方法。代码如下：

```
class GameEngine {
//…
bool running{ true };                              //游戏是否正在运行的标志
public:
    GameEngine(const int w = 50,const int h = 50) {}
    void run() {
        while (running) {
            processEvent();
            update();
                collision();
                render();
        }
        quit();
    }

    void processEvent() {}
    void update() {}
    void collision() {}
    void render() {}
    void quit(){}
};
```

用一个 GameEngine 类来表示游戏，一旦创建了一个表示具体游戏的 GameEngine 类

对象，就可以调用其 run()方法运行这个游戏：

```
int main() {
    GameEngine game;

    game.run();
}
```

7.12.2 贪吃蛇游戏

当然，对于一个具体的游戏，需要修改 GameEngine 类，添加针对特定游戏的一些特定属性，如贪吃蛇游戏可以包含表示游戏画布、蛇和鸡蛋 3 个对象。

```
class GameEngine {
    Window * window{nullptr};                      //游戏画布
    Snake * snake{ nullptr };                      //蛇
    BackGround bg;                                 //背景
    Egg * egg{ nullptr };                          //鸡蛋
    bool running{ true };                          //游戏是否运行标志
public:
    GameEngine(const int w = 50,const int h = 50){}
    //…
};
```

因此，还要定义表示游戏画布、背景、蛇和鸡蛋的类 Canvas、BackGround、Snake、Egg。

1. 窗口类 Window

可以将前面的帧缓冲器的相关数据及函数封装为一个 Window 类。

```
#include <iostream>
using Color = unsigned char;                       //用字符表示颜色

class Window {
    int width{ 60 }, height{ 50 };                 //窗口
    Color bg_color{ ' ' };                         //背景颜色用空格字符表示
    Color * frame_buffer{ nullptr };               //帧缓存,彩色图像的显示器内存
public:
    Window(int w, int h, Color bgColor)            //构造一个窗口对象
        :width{ w }, height{ h }, bg_color{ bgColor },
        frame_buffer{ new Color[w * h] }{}
    ~Window() {
        delete[] frame_buffer;                     //删除动态内存
    }
    //绘制一个(x,y)处的像素,即给该像素一个颜色 Color
    void set_pixel(int x, int y, Color color) {
        auto k = y * width + x;
        frame_buffer[k] = color;
    }
```

```
    //查询(x,y)处像素的颜色
    Color get_pixel(const int x, const int y) const {
        auto k = y * width + x;
        return frame_buffer[k];
    }

    //---------- 清空窗口 ---------
    void clear() {
        if (!frame_buffer) return;
        auto n = width * height;
        for (auto i = 0; i != n; i++)
            frame_buffer[i] = bg_color;            //设置该像素为背景颜色
    }

    //--------- 显示窗口的内容
    void show() {
        for (auto y = 0, k = 0; y < height; y++) {
            for (auto x = 0; x < width; x++, k++)
                std::cout << frame_buffer[k];
            std::cout << '\n';
        }
    }

    int get_width() { return width; }
    int get_height() { return height; }
    Color get_bg_color() { return bg_color; }
};
```

其中，frame_buffer 表示字符像素矩阵的动态内存的指针，即帧缓冲器。set_pixel()和 get_pixel()分别用来设置和查询像素的颜色。clear ()将帧缓冲器的所有像素的颜色设置为背景颜色 bg_color。show()用于在屏幕上显示出帧缓冲器中的所有像素。

在 GameEngine 的构造函数里创建一个动态的 Window 对象，并让其成员变量 window 指向这个对象(window = new Window (w, h, ' ');)，并修改 GameEngine 的成员函数 render()。

```
class GameEngine {
    Window * window{ nullptr };
public:
    GameEngine(const int w = 50, const int h = 40) {
        window = new Window(w, h, ' ');
        hideCursor();
    }
    ~GameEngine() { delete window; }
    //…
    void render() {
        gotoxy(0, 0);
        window->clear();                   //清空窗口
        draw_scene();                      //绘制场景
```

```
        window->show();                    //显示图像
    }
    void draw_scene() {      }
    //…
};
```

render()用于绘制和显示场景，其中调用了专门绘制游戏场景的辅助函数 draw_scene()，因为还没添加游戏场景数据，因此，该函数暂时是空的。

当再次运行前面的 main()函数时，将出现一个黑色的控制台窗口。

2. 游戏背景 BackGround

为了看到游戏的窗口，增加一个专门绘制游戏背景的 BackGround 类，在贪吃蛇游戏中，该类的 draw()方法只是简单地在窗口上绘制游戏窗口的边框。

```
class BackGround {
    Color top_boundary_color{ ' ' }, bottom_boundary_color{ '_' };
    Color left_right_boundary_color{ '|' };
public:
    void draw(Window &window) {
        auto right{ window.get_width() - 1 };
        auto bottom{ window.get_height() - 1 };
        for (auto x = 0; x < window.get_width(); x++) {
            window.set_pixel(x, 0, top_boundary_color);
            window.set_pixel(x, bottom, bottom_boundary_color);
        }

        for (auto y = 0; y < window.get_height(); y++) {
            window.set_pixel(0, y, left_right_boundary_color);
            window.set_pixel(right, y, left_right_boundary_color);
        }
    }
};
```

在 GameEngine 类的场景绘制函数 draw_scene()中添加绘制背景的代码：

```
void draw_scene() {
    bg.draw(*window);
}
```

当再次运行前面的 main()函数时，将出现一个包含背景的游戏窗口(见图 7-14(a))。

3. 鸡蛋 Egg

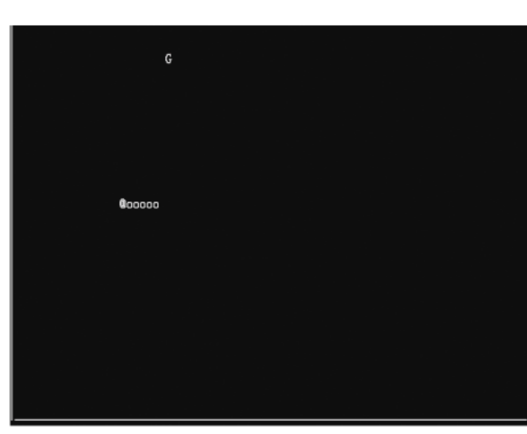
定义一个 Egg 类来表示一个鸡蛋，鸡蛋有位置、大小、颜色和绘制形状的功能。

```
class Egg {
    int x, y;           //鸡蛋位置
    int size{ 1 };      //鸡蛋大小
    Color color;        //鸡蛋颜色
```

```
public:
    Egg(int x, int y, Color color = 'G', int s = 1) :x{ x }, y{ y },
        size{ s }, color(color){}
    void draw(Window& window) {     //在 window 表示的窗口画布上绘制鸡蛋形状
        window.set_pixel(x, y, color);
    }
    Color get_color() { return color; }
};
```



(a) 游戏窗口及背景



(b) 蛇和鸡蛋

图 7-14　游戏窗口及背景、蛇和鸡蛋

4. 蛇 Snake

在控制台游戏中，可以将蛇看成一系列字符像素的线性表。每个字符像素都有一个位置，当蛇在用户控制下运动时，蛇头像素沿着控制方向前进一个像素，带动其他像素跟着移动。

如图 7-15 所示，蛇身的每个像素都要移动一个位置，即修改每个像素的位置。当蛇身变得很长时，每次移动都需要修改所有的蛇身像素位置，这是比较耗时的。仔细分析这个运动过程，可以发现一个规律：除蛇头像素外，每个蛇身像素实际上移动到其前一个蛇身像素的位置，如蛇尾像素移动到蛇尾前一个像素的位置，而这些蛇身像素都是一样的。因此，蛇的每次移动可用 2 步操作来模拟：

```
OOOOOOOO@
```

(a) 移动前

```
 OOOOOOOO
        @
```

(b) 移动后

图 7-15　蛇的前进

(1) 在蛇头前进位置插入一个蛇头像素成为新的蛇头并修改原来的蛇头像素为蛇身像素。

(2) 删除原来的蛇尾像素。

通过这 2 步操作就相当于蛇前进了一个位置。

因此，只需要在蛇头插入、在蛇尾删除一个像素，如果用线性表存储蛇的所有像素，只要在线性表的一端插入、另一端删除一个像素，从而提高了程序效率。

为了避免插入、删除移动大量元素，这里可以用链表来存储蛇身的所有像素。

首先定义一个表示位置的类 Position：

```
//表示一个位置
class Position {
    int x{ 0 }, y{ 0 };
public:
    Position(int x = 0, int y = 0) :x{ x }, y{ y }{}
    void set_x(int x) { this->x = x; }
    void set_y(int y) { this->y = y; }
    auto get_x() { return x; }
    auto get_y() { return y; }
};
```

然后定义一个链表的结点 SnakeNode,用来表示一个蛇身像素,这个结点除了蛇身像素的位置 Positon 变量外,还有一个 next 指针变量指向下一个蛇身像素结点:

```
//一个蛇身像素在内存中的结点表示
class SnakeNode{
    Position pos{};                        //蛇身像素位置
    SnakeNode * next{nullptr};             //下一个蛇身像素结点的指针
public:
    SnakeNode(const Position pos, SnakeNode * n = nullptr)
        :pos{pos}, next{ n }{}
    Position get_pos() { return pos; }
    SnakeNode * get_next() { return next; }  //返回该结点的 next 值,即指向下一个结点的指针
    void set_next(SnakeNode * n) { next = n; }   //修改 next 值
};
```

蛇身可以分别用 2 个指针表示:链表头结点和尾结点的指针,分别用来表示蛇尾和蛇头。

蛇创建时其初始位置是随机的,但具有一定的长度。蛇还有一个前进方向,初始时,其前进方向就是蛇身的方向。此外,蛇还可以吃蛋,因此,可以定义如下 Snake 类:

```
class Snake {
    //蛇身用 2 个结点指针变量分别指向链表的头结点和尾结点.
    SnakeNode *head{ nullptr }, *tail{ nullptr };
    int direction{};                //蛇前进方向
    Color body_color, head_color;
    bool idead{ false };            //蛇是否死亡
    int width{ 0 }, height{ 0 };
    bool eating{ false };
public:
    //初始化窗口范围[width,height]指定长度的一条蛇
    Snake(const int width, const int height, int length = 3,
        Color body_color = 'o', Color head_color = '@');

    void draw(Window& window);          //在 window 画布上绘制自己的形状

    //沿给定方向前进,前进过程中需要检查是否发生了碰撞
    void move(char direction);
```

```
    void eat(bool eating);                    //吃了一个鸡蛋
    void set_direction(int d) {               //设置蛇的运动方向
        direction = d;
    }
    SnakeNode * get_head() { return head; }   //返回链表的头结点
    SnakeNode * get_tail() { return tail; }   //返回链表的尾结点
    Color get_body_color() { return body_color; }
    Color get_head_color() { return head_color; }
};
```

Snake的构造函数用于初始化窗口范围[width,height]指定长度的一条蛇，并且包含了表示蛇身和蛇头的颜色body_color和head_color。move(char direction)函数沿指定方向direction移动一个位置。eat()吃一个蛋，会使蛇身变长(增加一个像素)。

首先实现Snake的初始化构造函数。

```
Snake::Snake(const int width, const int height, int length,
Color body_color, Color head_color) {
    this->width = width;
    this->height = height;
    this->body_color = body_color;
    this->head_color = head_color;

    //生成随机的蛇的位置
    auto x_min{ length + 1 }, x_max{ width - x_min },
        y_min{ length + 1 }, y_max{ height - y_min };
    auto x = random_int(x_min, x_max);
    auto y = random_int(y_min, y_max);

    SnakeNode *p = new SnakeNode(Position(x, y));  //创建蛇头结点
    tail = p;                                      //该结点是链表的尾结点(最后的结点)
    head = new SnakeNode(Position(), p);           //创建整个链表的头结点
    auto d = random_int(0, 4);                     //生成随机的 0,1,2,3
    for (auto i = 1; i != length; i++) {           //生成其他的蛇身结点
        if (d == 0) x++;
        else if (d == 1) x--;
        else if (d == 2) y++;
        else y--;
        p = new SnakeNode(Position(x, y), head->get_next());
        head->set_next(p);
    }
}
```

其中用一个辅助函数random_int()生成[x_min,x_max]的一个随机整数，用于生成蛇头的随机位置。

```
#include <cstdlib>
#include <ctime>
inline int random_int(const int x_min, const int x_max) {
    static bool is_seeded = false;
```

```
    if (!is_seeded) {
        srand((unsigned)time(0));                //生成随机数种子
        is_seeded = true;
    }
    return rand() % (x_max - x_min) + x_min;
}
```

有了蛇的初始位置，就可以创建一个表示蛇头像素的结点：

```
SnakeNode *p = new SnakeNode(Position(y, x));       //创建蛇头结点
```

结点的地址保存在指针变量 p 中，然后创建一个链表的头结点，p 的值作为它的 next 指针变量的值。即头结点的 next 指向这个 p 指向的结点，如图 7-16(a)所示。

接着生成代表蛇前进方向的随机方向 d(d = random_int(0, 4))，根据 d 不断计算下一个蛇身像素的位置，并用这个位置创建一个新的结点。同时将这个结点插入在头结点的后面：

```
p = new SnakeNode(Position(x, y),head->next);
head->set_next(p);
```

将 p 指向的这个结点采用“前插法”插入到链表中，即 p 指向的新结点的 next 就是 head—>next，也即它们都指向原来的头结点后的那个结点；接着修改 head 指向结点的 next 变量值为 p(head—>next = p;)，即指向这个新的结点，从而新结点就插入在头结点之后，成为新的首结点。对于每个蛇身像素结点都重复这个过程，得到具有一定长度的蛇，如图 7-16(b)所示。

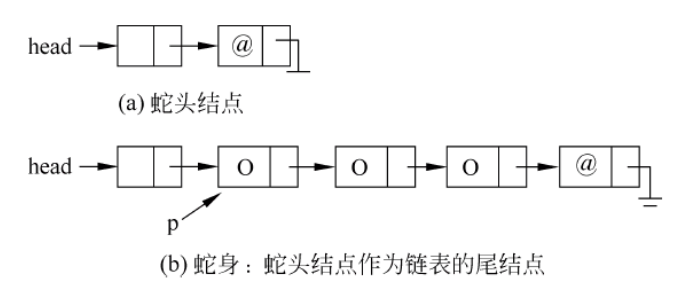


图 7-16　初始化蛇身

注意：为了便于实现蛇的 move()和 eat()功能，规定头结点后的首结点表示蛇尾，而链表的尾结点表示的是蛇头，即链表头表示蛇尾、链表尾表示蛇头。

创建一个蛇后，就可以让它在画布上绘制自己。

```
void Snake::draw(Window& window){
    SnakeNode *p = head->get_next();
    while (p != tail) {                              //遍历每个蛇身结点
        window.set_pixel(p->get_pos().get_x(),
            p->get_pos().get_y(), body_color);       //在画布中设置这个蛇身像素结点的
                                                     //颜色
        p = p->get_next();                           //指针 p 向后移动，指向下一个结点
```

```
    }
    window.set_pixel(p->get_pos().get_x(),          //在画布上绘制蛇头结点
        p->get_pos().get_y(), head_color);
}
```

这里,用一个指针变量 p 开始指向链表头结点后的那个结点,如图 7-16(b)所示,只要 p 不是尾结点(蛇头)就用蛇身颜色 body_color 绘制这个结点,并将 p 移向下一个结点(p = p—> get_next();),退出循环后,用蛇头颜色绘制蛇头结点。

现在在 GameEngine 中添加表示蛇和蛋的成员变量,修改 GameEngine 的构造函数,添加创建随机出现的蛇 Snake 和鸡蛋 Egg 对象的代码。

```
class GameEngine {
    Window * window{ nullptr };
    bool running{ true };                       //游戏是否正在运行的标志
    bool start{ false };                        //游戏是否开始
    BackGround bg;
    Snake * snake{ nullptr };
    Egg * egg{ nullptr };
public:
    GameEngine(const int w = 60, const int h = 50) {
        window = new Window(w, h, ' ');
        //创建 Snake 对象
        snake = new Snake(w, h, 4);             //构造一个位置随机的蛇 Snake 对象
        //创建随机位置的 Egg 对象
        auto x = random_int(2, w - 2);
        auto y = random_int(2, h - 2);
        egg = new Egg(x, y);
    }
    ～GameEngine() {
        delete window; delete snake;delete egg;
        }
    //…
}
```

修改 GameEngine 类的的 draw_scene ()函数,添加绘制蛇和鸡蛋的代码:

```
void draw_scene() {
    bg.draw( * window);
    if(snake)snake->draw( * window);            //让蛇绘制自己
    if (egg)egg->draw( * window);               //让鸡蛋绘制自己
}
```

再次运行前面的 main()函数,将出现一个初始的蛇和一个鸡蛋。

Snake 的 eat()函数就是简单设置一下 eating 标志,表示是否正吃鸡蛋。

```
void Snake::eat(bool eating) {
    this->eating = eating;
}
```

move()成员函数根据前进方向使得蛇头前进一个位置，代码如下：

```
void Snake::move() {
    auto head_pos = tail->get_pos();     //当前蛇头位置，注意：链表尾部表示蛇头
    //根据前进方向，确定新的蛇头位置
    auto x{ head_pos.get_x() }, y{ head_pos.get_y() };
    if (direction == 0)         //上键，向上移动，--y
        --y;
    else if (direction == 1)    //下键，向下移动，++y
        ++y;
    else if (direction == 2)    //左键，向左移动，--x
        --x;
    else                        //右键，向右移动，++x
        ++x;

    //创建新的蛇头，加入到链表的尾部
    SnakeNode *p = new SnakeNode(Position(x,y));    //创建新的结点
    tail->set_next(p);          //p加到尾结点(tail)的后面，即蛇头结点的后面
    tail = p;                   //p成为新的链表尾结点，即p成为新蛇头结点
    //如果没有吃鸡蛋，则删除蛇尾结点
    if (!eating) {
        //删除代表蛇尾的链表首结点(头结点后的那个结点)
        p = head->get_next();   //p指向首结点
        head->set_next(p->get_next());   //p的next结点成为head的后一个结点
        delete p;               //释放p结点占用的内存
    }
    //否则，正吃了一个鸡蛋，不用删除蛇尾结点，相当于增加了一节蛇尾，但应清空吃蛋标志
    else eating = false;        //鸡蛋已经吃完
}
```

先得到当前蛇头的位置，再根据前进方向确定蛇头的新位置(x,y)，在这个位置创建代表新蛇头的链表结点，并加到链表的最后面(因为链表尾表示蛇头)。

如果没有遇到鸡蛋(eating 为 false)，就删除蛇尾结点(即链表的头结点后的那个首结点)，表示整个蛇身前进了。如果正吃了一个鸡蛋，则不删除蛇尾结点，相当于增加了一节蛇尾。

删除蛇尾结点就是删除链表的首结点的过程(即前面链表的 pop_front()函数)，即先将它保存到临时变量 p 中(即 p = head—> get_next();)，然后修改 head 指向的头结点中的 next 指针变量设置为 p 指向结点的 next 指针值(即 head—> set_next(p—> get_next());)，这样就将 p 指向的结点从链表中断开了，最后释放 p 指向的结点的内存，防止内存泄漏。

游戏开始时，蛇是不动的，当用户按下某个键如空格键时，游戏开始，蛇开始运动。为此，需要在事件处理函数 processEvent()中检测用户按键。

```
void processEvent() {           //处理事件
    //处理事件
    char key;
    if (_kbhit()) {
        key = _getch();
```

```
            if (key == 27) running = false;
            else if (key == ' ') start = !start;
        }
    }
```

即用空格键改变游戏的开始标志 start，然后修改一下函数，当游戏处于开始状态时，每次 update()都让蛇移动一个位置。

```
void update() { if(start) snake->move(); }
```

运行这个程序，当按下空格键后，蛇就沿固定方向自己运动了，直到跑出窗口。

用户可以通过按键来控制蛇的运动方向，如用上、下、左、右箭头键控制蛇的运动，只要在 processEvent()里添加这些按键处理功能，使得根据不同的按键，调整蛇的前进方向(direction)。

```
void processEvent() {
    //处理事件
    char key;
    if (_kbhit()) {
        key = _getch();
        if (key == 27) running = false;
        else if (key == ' ') start = !start;
        else {
            start = true;
            if (key == KEY_UP)
                snake->set_direction(0);
            else if (key == KEY_DOWN)
                snake->set_direction(1);
            else if (key == KEY_LEFT)
                snake->set_direction(2);
            else if (key == KEY_RIGHT)
                snake->set_direction(3);
        }
    }
}
```

当再次运行前面的 main()函数时，就可以用上、下、左、右箭头键控制蛇的运动方向。

然而，可以发现 2 个明显的问题：

(1) 当按键使新的前进方向和原来前进方向正好相反时，蛇会沿着身体的反方向运动，导致蛇的像素结点发生了重叠。

(2) 蛇身体会移出画面窗口，因其坐标超出画布的范围，在绘制时会导致程序崩溃。

第 1 个问题很好解决，只要禁止新前进方向和原前进方向相反就可以了。即修改 Snake 的成员函数 set_direction()。

```
void set_direction(int d) {        //设置蛇的运动方向
    if(d== 0 && direction == 1 ||
        d == 1 && direction == 0 ||
```

```
        d == 2 && direction == 3 ||
        d == 3 && direction == 3) return;
    direction = d;
}
```

当蛇和窗口发生碰撞或蛇自身发生碰撞都导致蛇的死亡、程序结束。还有一种是蛇和鸡蛋碰撞，这个时候就可以调用蛇的 eat()函数使蛇身变长。

这些碰撞检测处理由 GameEngine 的 collision()函数完成，因为这是一个简单的游戏，画面上的颜色也很简单，可以采用一个简单技巧检测碰撞，即在开始前进到新蛇头位置时，检查这个位置上的是什么物体（是墙、蛇身、还是鸡蛋）来检查是否碰撞以及碰撞的类型。代码如下：

```
void collision() {
    if (!start) return;
    auto tail = snake->get_tail();
    auto pos = tail->get_pos();                     //蛇头位置
    auto x{ pos.get_x() }, y{ pos.get_y() };

    if (x == 0 || y == 0 || x == window->get_width() - 1
        || y == window->get_height() - 1) {
        running = false;                            //超出窗口,蛇死亡,游戏结束
        return;
    }

    Color color = window->get_pixel(x, y);          //得到该位置的颜色
    if (color == window->get_bg_color()) return;    //未发生碰撞

    if (color != snake->get_head_color()) {
        if (egg&&color == egg->get_color()) {       //遇到了鸡蛋
            snake->eat(true);                       //蛇吃了鸡蛋
            auto x = random_int(2, window->get_width() - 2);
            auto y = random_int(2, window->get_height() - 2);
            delete egg;  egg = new Egg(x, y);       //销毁鸡蛋,创建新鸡蛋
        }
        else {                                      //和墙或自身发生碰撞,游戏结束
            running = false;                        //和墙或自身发生碰撞,游戏结束
            return;
        }
    }
}
```

当蛇头位置超出窗口时，游戏结束。如果蛇头位置的颜色不是蛇头颜色，表示蛇头前进到了一个新位置，则根据其是否是鸡蛋的颜色确定遇到了鸡蛋还是碰到了墙或自身而分别处理。

当然，也可以直接用蛇头和蛇身的每个位置、窗口边框进行直接的碰撞检测，作为练习，读者可以重写上述的碰撞检测代码。

同时，为了防止绘制超出画面的蛇身像素，在 render()函数的最前面添加如下代码：

```
void render() {
        if (!running) return;                            //游戏未运行或结束时不显示画布
   //…
}
```

再运行 main()函数，这时吃鸡蛋就能使蛇身变长了，如图 7-17 所示。

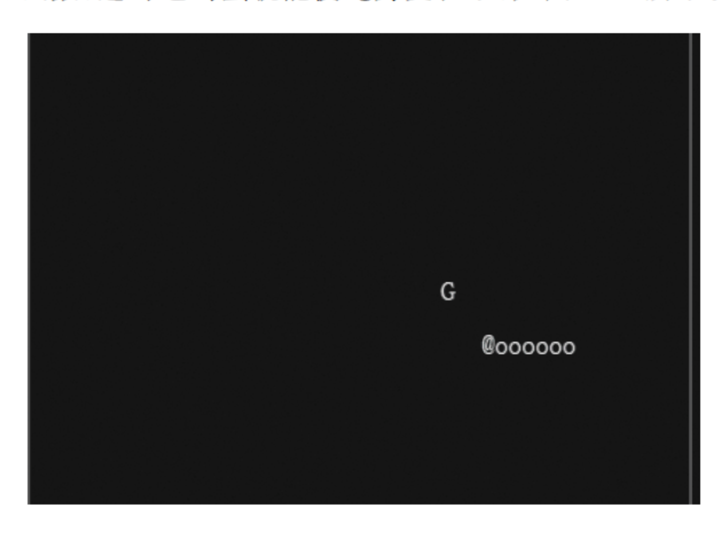


图 7-17　贪吃蛇游戏运行画面

5. 问题

目前的贪吃蛇游戏还有一些问题：

(1) 缺少速度控制。开始时速度应该比较慢，随着蛇身变长，说明用户操作越来越熟练，这个时候速度应该变快，还应给不同级别用户设置不同的蛇的运动速度。

(2) 当蛇吃到鸡蛋时，新的鸡蛋应该距离蛇身体有一定距离。

7.13 习题

1. 使用 struct 和 class 定义类有什么区别？
2. 类中的 this 指针表示什么？
3. 什么叫内联成员函数？内联成员函数有什么优点？如何定义一个内联成员函数？什么样的函数适合定义为内联成员函数？
4. explicit 关键字的作用是什么？请举例说明其用法。
5. 什么叫委托构造函数？请举例说明。
6. 下列代码的错误是什么？

```
struct X{
    void print(int i = 3);
};
void X::print(int i = 3){ }
```

7. 下列代码有什么错误？原因是什么？如何修改？

```
class Point2 {
    int x,y;
    static inline const  Point2 O{0, 0};
public:
    Point2(const int x = 0.,const int y = 0.):x(x),y(y) {};
};
```

8. 下面程序的输出是什么？

```
#include <iostream>
using namespace std;
class X {
    static int x;
    int * ptr;
    int y;
};
int main() {
    X t;
    cout << sizeof(t) << " ";
    cout << sizeof(X *);
}
```

9. 下面程序的输出是什么？

```
class X {
public:
    X() { std::cout << "1"; }
    X(const X& other) { std::cout << "2"; }
    X & operator = (const X & other) { std::cout << "3"; return * this; }
};

int main() {
    X  x, y = x;
}
```

10. 下面程序的输出是什么？

```
class X{
public:
    int i;
    void get();
};
void X::get(){
    std::cout << "输入 i: ";
    std::cin >> i;
}
X t;                                    //全局对象
```

```
int main(){
    X t;                                          //局部对象
    t.get();
    std::cout << "局部变量 t 的 i 值是: " << t.i << '\n';
    ::t.get();                                    //::是"全局作用域"限定符
    std::cout << "全局变量 t 的 i 值是: " << ::t.i << '\n';
    return 0;
}
```

11. 下面程序的输出是什么?

```
#include <iostream>
struct A {
    A() { std::cout << "A"; }
    A(const A &a) { std::cout << "B"; }
    virtual void f() { std::cout << "C"; }
};
int main() {
    A a[2];
    for (auto x : a) {
        x.f();
    }
}
```

12. 补充?处的代码,运行 main(),观察构造函数和析构函数的调用情况,如 P1 和 P2 的销毁的次序和创建次序是相反的,并说明一共调用了多少次 Point 类的成员函数。

```
#include <iostream>
using namespace std;

class Point{
    int x{ 0 }, y{ 0 };
public:
    Point(){
        cout << "Constructor Called"<< endl;
    }
    Point(int X, int Y = 20){
        //?
        cout << "Constructor Called" << x <<","<< y << endl;
    }
    ~Point(){
        cout << endl << "Destructor Called" << x << "," << y << endl;
    }
    void set_x(const int x) {?}
    void print()    {
        cout << x << "," << y << endl;
    }
};
int main(){
    Point p1 = Point(10);
```

```
    cout << "p1 Value Are : ";
    p1.print();
    Point p2 = Point(30, 40);
    cout << "p2 Value Are : ";
    p2.print();
    p1.set_x(2);
    p1.print();
    return 0;
}
```

13. 为12题的Point类添加一个友元函数 Point add(Point p,Point q),用于2个Point对象的相加运算。

14. 为12题的Point添加一个静态计数器变量count,用来记录从Point类实例化的对象的个数,并添加一个静态函数clear()用于将count设置为0。

15. 实现一个表示时间"时:分:秒"的Time类,可以用"时:分:秒"格式的字符串构造一个Time对象,可以用get_second()返回总秒数。

```
#include <iostream>
using namespace std;
class Time{
    int hours.minutes.seconds;
public:
    Time(int h, int m, int s);
    Time(const char * str);          //str 是字符串格式的时间"02: 08: 15"
    void getTime(void);
    void putTime(void);
    void addTime(Time T1, Time T2);
    int seconds();                   //将时间转换为秒
};
```

16. 实现下面的表示长方体的类Box,并编写代码测试这个类的功能。

```
class Box{
private:
    double length{ 1.0 };
    double width{ 1.0 };
    double height{ 1.0 };
    static inline size_t objectCount{}; //Box 对象的个数
public:
    //Constructors
    Box(double lv, double wv, double hv);
    Box(double side);                   //构造一个边长为 side 的长方体
    Box();                              //默认构造函数
    Box(const Box& box);                //拷贝构造函数
    Box& operator = (const Box& box);   //赋值运算符函数
    double volume() const;              //体积计算函数
    size_t getObjectCount() const { return objectCount; }
};
```

17. 定义一个表示字符串的类 String，其数据成员是指向 C 风格字符串的指针。

```
class String {
    char * s{nullptr};
public:
    String();
    String(const char * str);
    char get_ch(const int i)const;
    bool set_ch(const int i,const char ch);
};
```

18. 结合例子说明静态成员和非静态成员的区别。

19. 分别给表示顺序表和链表的类添加一个逆置成员函数 converse()，用于将元素顺序逆向排列，即将(a_1,a_2,…,a_n)转变为(a_n,a_{n-1},…,a_1)。

20. 为链表类添加下列的成员函数并测试它们。

ElemType front()：查询链表第一个元素的值。

ElemType back()：查询链表最后一个元素的值。

bool push_back(constElemType e)：在链表尾部添加一个元素。

bool pop_back()：删除链表最后一个元素。

bool Next((ElemType &e)：返回当前元素，并将内部的指针向后移动一个位置。

bool First(ElemType &e)：内部指针定位第一个元素并返回第一个元素。

int find(const ElemType e)：查询是否存在等于 e 的元素，返回其序号。

21. 模仿图书管理程序，编写一个学生成绩管理程序，用一个类描述一个学生的信息，每个学生的信息包含姓名、学号、平时成绩、实验成绩、期末成绩、总评成绩。其中总评成绩不需要输出，根据用户输入的分数比例从其他 3 个成绩中自动计算，并统计不及格($s<60$)、及格($60\leqslant s<70$)、中等($70\leqslant s<80$)、良好($80\leqslant s<00$)、优秀($s>90$)的百分比。所有学生数据用一个线性表(顺序表或链表)存储。

22. 根据你的经验和看法改进并完善贪吃蛇游戏。



运算符重载

任何复杂的程序最终都归结为运算符对基本类型的运算，即表达式是构成程序的最基本的计算单元。对于基本类型的变量，运算符的含义总是相同的，例如加法运算符＋对 int、float、double 等都具有同样的含义。

用人们熟悉的运算符对数据进行运算，具有简洁、直观的优点。例如 add(x, multiply(y, z))显然没有 x＋y＊z 直观易懂。C++的运算符不仅可用于基本类型的运算，还允许程序员通过**运算符重载**(**operator overloading**)的方式使得运算符可以用于用户定义类型对象的运算。

例如，对于用户定义类型 string，可以用＋运算符将两个字符串拼接为一个更大的字符串，可以用赋值运算符＝将一个字符串对象赋值给另外一个字符串对象，可以用输出流运算符<<将一个字符串对象输出到控制台窗口上：

```
std::string s1{ "hello" }, s2{ "world" }, s3;
s3 = s1 + s2;                    //用 + 、= 对 string 对象进行运算符
std::cout << s3;                 //用<<将 s3 输出到 cout 中
```

对于用户定义的类型，只有重新定义了某个运算符的工作方式，才能将这个运算符用于这种类型的对象。

运算符重载其实就是普通的函数重载，在 C++中，每个运算符实际就是一个函数，称为**运算符函数**。运算符函数的完整名称是 **operator** 关键字加上运算符，例如，加法运算符＋的函数名是 **operator＋**()，赋值运算符＝的函数名是 **operator＝**()，输出运算符<<的函数名是 **operator** <<()。

重新定义针对用户定义类型的运算符函数，称为**运算符重载**。在 7.3.4 节，对 Date 类型，重载了赋值运算符＝，使得可以用“＝”对 2 个 Date 对象进行赋值运算。

8.1 运算符重载的 2 种方式

对于一个用户定义类型，运算符重载的方式分为以下 2 种。

- 成员函数：将运算符函数定义为类的成员函数。

• 外部函数：将运算符函数定义为外部函数(全局函数)。

大多数运算符可以采用上述任意一种重载方式。但某些特殊运算符只能采用其中一种方式重载。例如，赋值运算符只能作为类的成员函数重载。

下面的 Point 类表示的是二维屏幕上的一个点，该类以成员函数的方式重载定义了加法运算符函数 **operator+**()。

```
class Point {
    double x{}, y{};
public:
    Point(double x, double y) :x{ x }, y{ y }{}
    Point operator + (const Point &other){
        return  Point(x + other.x, y + other.y);
    }

    friend void print(const Point &p);
};
void print(const Point &p) {
    std::cout << p.x << "," << p.y;
}
```

然后就可以使用运算符+，对 2 个 Point 类对象进行加法运算：

```
int main() {
    Point P{ 2,3 }, Q{ 4,5 };
    print(P + Q);
}
```

P+Q 实际上是 P. **operator+**(Q)的简写形式。即通过对象 P 调用 Point 的成员函数 operator+()并将参数 Q 传递给这个函数。

注意："+"是一个二元运算符，如果作为成员函数实现时，这个函数只能带一个参数而不能带 2 个参数，即不能将成员函数写成"Point operator+(Point p1,Point p2)"这种形式。因为调用这个函数的对象就是第一个操作数(左操作数)。在成员函数中，有一个隐含的 this 指针就指向这个调用的对象(this 存储的是这个对象的地址)。上述的 operator+()成员函数实际上是：

```
Point operator + (const Point & other) {
    return  Point(this->x + other.x, this->y + other.y);
}
```

即通过 this 指针访问调用这个函数的对象的 x 或 y。

因此，如果用成员函数方式重载一个二元运算符@，该成员函数只能有一个参数(表示的是右操作数)，假如 a、b 是这种类的对象，可以解释为：a. **operator@**(b)。

也可以用外部函数的方式重载刚才的加法运算符+。

```
class Point {
    …
    friend Point operator + (Point P, const Point&other);
  …
};
Point operator + (Point P, const Point&other) {
    return  Point(P.x + other.x, P.y + other.y);
}
```

同样,可以用于 Point 对象的相加:

```
int main() {
    Point P{ 2,3 }, Q{ 4,5 };
    print(P + Q);
}
```

P + Q 实际上调用的是 operator+(Point P, const Point&other),也就是普通的外部函数。

上述代码中,为了能访问 Point 变量的私有变量,这个外部运算符函数 operator+()在 Point 类中被声明为 Point 的友元。

注意:作为外部函数重载一个二元运算符@,运算符必须有 2 个参数,不能少于或多于 2 个参数(表示该运算的 2 个操作数)。即形如:

```
T opertor@(const T1 & a, const T2 & b)
```

对于 T_1 类型的对象 a 和 T_2 类型的对象 b,a+b 就是 **operator@**(a,b)。

同样,对于一个一元运算符@,如果作为类的成员函数,则不能有任何参数,因为第一个参数就是调用这个运算符的对象自己。如果作为外部函数重载,则带有一个参数。例如,如果将一元的负号运算符-作为 Point 类的成员函数:

```
#include <iostream>
class Point {
    double x{}, y{};
public:
    Point(double x, double y) :x{ x }, y{ y }{}
    Point operator - ()const {
        return Point( - this -> y, - this -> x);
    }
    void print() { std::cout << x << "," << y; }
    //...
};
```

如果作为外部函数,则应如下实现:

```
class Point {
    double x{}, y{};
```

```
public:
    Point(double x, double y) :x{ x }, y{ y }{}
    friend Point operator - (const Point &p);
    void print() { std::cout << x << "," << y; }
    //...
};
Point operator - (const Point &p) {
    return Point( - p.y, - p.x);
}
```

然后，可以用这个一元的负号运算符－：

```
int main() {
    Point p(3, 4);
    ( - p).print();
}
```

程序输出：

```
-4, -3
```

注意：这里实现的 operator－()是作为一元运算符的负号运算符而不是作为二元运算符的减法运算符。

运算符作为成员函数和外部函数重载的主要区别如下：作为成员函数重载的运算符的第一个操作数必须是这个类的对象，不能是其他可转换为这个类类型的变量；而作为外部函数重载的运算符的第一个操作数可以是能转换为这个类类型的变量。

例如，假如 Point 有一个带一个 double 类型参数的构造函数，这个构造函数就定义了一个类型转换，即可以将一个 double 类型的值转换为一个 Point 类的值。

```
class Point {
    double x{}, y{};
public:
    Point(double x) :x{ x }, y{ 0 } {}
    //...
};
```

如果运算符 operator＋是作为外部函数重载的，则：

```
P + 2;
2 + P;
P + Q;
```

上述 3 个表达式语句都是正确的，因为 2 会被自动转换为 double，然后从 double 自动转换为 Point，最后调用 operator＋()对 2 个 Point 对象相加，如 2＋P 被转换为 operator＋(Point(2)，P)。

如果运算符 operator＋是作为 Point 的成员函数重载的，则 2＋P 就会产生编译错误，

因为 2 本身不是一个 Point 对象，编译器不会这样操作：Point(2).operator(P)。这是错误的。因此，作为成员函数重载的运算符的第一个操作数（运算数）必须本身就是类的对象而不能是类型转换后的对象。该规则对于一元运算符也是一样的。

8.2 赋值运算符=

对于 Point 对象，可以直接用赋值运算符=：

```
P = Q;
```

为什么没有重载赋值运算符=就能直接使用赋值运算符=？前面说过，C++编译器会自动生成一个默认的赋值运算符成员函数 operator=()。对于 Point 类，这个默认的赋值运算符就足够了，但如果一个类包含有申请一些资源的成员变量，如指向动态内存的指针变量等，默认的赋值运算符就不适用了，需要程序员重新定义这个类的赋值运算符函数 operator()=。

对于 Point 类，编译器生成的默认赋值运算符 operator=()如下：

```
Point & operator = (const Point & other) {
    if (this != &other) {
        x = other.x; y = other.y;
    }
    return * this;
}
```

7.10 节的 String 类用一个成员变量 data 指向动态分配的内存，就需要重新定义赋值运算符函数 operator=()。如果不这样做，默认的赋值运算符会使得被赋值对象和原来的对象的 data 成员变量共享同一块动态内存块，当这 2 个对象释放时都会释放这块内存，导致"多次释放同一块内存"的致命错误。

注意：只能以成员函数的形式重载赋值运算符 operator=()，并且重载的函数最后必须返回自引用(* this)。

8.3 下标运算符[]

许多对象的数据不是一个单一值而是多个值，如上面的 Point 里有 2 个 double 类型的值分别表示一个点的 x 和 y 坐标。可以通过下标运算符[]，给每个值对应唯一的一个下标，然后通过下标运算符[]访问相应的值。这就需要对该类重载下标运算符 operator[]函数。

```
class Point {
    double x{}, y{};
public:
    Point(double x, double y) :x{ x }, y{ y }{}
    double& operator[](int i) {                    //返回对象的引用
```

```
        if (i == 0) return x;
        else if (i == 1) return y;
        else throw "下标非法";
    }
    double operator[](const int i) const{        //返回值的 const 函数
        if (i == 0) return x;
        else if (i == 1) return y;
        else throw "下标非法";
        }
    }
    …
}
```

下标运算符 operator[]函数根据下标参数是 0 或 1 返回 x 或 y 值，如果是其他下标，就用 **throw** 关键字抛出一个异常对象(关于异常，第 14 章会介绍)。

下标运算符通常定义 2 个版本：一个是返回可以被修改的引用，即可以作为赋值运算符的左操作数；另外一个是 const 成员函数，返回的是一个值，可用作赋值运算符的右操作数。如：

```
int main() {
    Point P{ 2,3 };
    P[0] = 4;             //P[0]调用的是引用版本，
    P[1] = P[0];          //P[1]调用的是引用版本，P[0]调用的是 const 版本
    print(P);
}
```

8.4 输入输出运算符

可以对用户定义类型重载输入输出运算符>>或<<，例如：

```
class Point {
    …
    friend std::ostream& operator <<(std::ostream &out, const Point &p);
    …
};
std::ostream& operator <<(std::ostream &out, const Point &p) {
    out << "(" << p.x << "," << p.y << ")";
}
```

对 Point 类重载了输出运算符 operator <<，其第一个参数是输出流对象的引用，然后在程序中就可以使用<<输出一个 Point 对象：

```
std::cout << P;
```

注意：其中的输出流参数 out 必须是引用。读者可以模仿上述代码重载输入运算符>>。

8.5 比较运算符

可以重载比较运算符,例如下面的代码重载了<和==运算符:

```
class Point {
    …
    bool operator <(const Point &other);
    bool operator == (const Point &other);
    …
};

bool Point::operator <(const Point &other) {
    if (x  ==  other.x) return y < other.y;
    return x < other.x;
}
bool Point::operator == (const Point &other) {
    return x  ==  other.x && y == other.y;
}
```

然后就可以对 2 个 Point 对象通用<或==进行比较:

```
int main() {
    Point P{ 2,3 },Q(3,2);
    if (P < Q || P == Q) std::cout << "P <= Q";
    else std::cout << "P > Q";
}
```

也可以将<或==作为外部函数来实现:

```
class Point {
    …
    friend bool operator <(const Point P, const Point &Q);
    friend bool operator == (const Point P, const Point &Q);
    …
};

bool operator <(const Point P, const Point &Q) {
    if (P.x  ==  Q.x)return P.y < Q.y;
    return P.x < Q.x;
}
bool operator == (const Point P, const Point &Q) {
    return P.x  ==  Q.x && P.y  ==  Q.y;
}
```

当然也可以重载其他的比较运算符,如>、!=、<=、>=。实际上,这些运算符不是独立的,只要实现了<和==运算符,其他的运算符如<=可以用<和==运算符来实现。例如:

```
bool Point::operator<=(const Point &other) {
    return *this<other || *this==other;
}
```

对每种类型都重复对所有比较运算符进行重载，有点单调、烦琐。C++标准库的头文件 utility 已经用**模板**(第 10 章会介绍模板)为任何类型定义好了这些比较运算符模板，只需要程序员对一个类型重载定义<和==运算符，utility 头文件中的其他比较运算符模板会自动生成针对该类型的其他比较运算符重载函数。

utility 中的这些比较运算符函数模板属于名字空间 std::rel_ops，因此需要包含头文件和引入名字空间 std::rel_ops：

```
#include<utility>
using namespace std::rel_ops;
```

例如对于 Point 类，只需重载<和==运算符：

```
#include<utility>
using namespace std::rel_ops;

class Point {
    double x{}, y{};
public:
        …
    friend bool operator<(const Point P, const Point &Q);
    friend bool operator==(const Point P, const Point &Q);
    };
bool operator<(const Point P, const Point &Q) {
    if (P.x == Q.x)return P.y<Q.y;
    return P.x<Q.x;
}
bool operator==(const Point P, const Point &Q) {
    return P.x == Q.x && P.y == Q.y;
}
```

利用 std::rel_ops 中的比较运算符模板就可以对 2 个 Point 对象使用任何比较运算进行比较：

```
int main() {
    Point P{ 2,3 },Q(3,2);
    if (P>Q) std::cout << "P>Q";        //>运算符
    if(P!=Q) std::cout << "P!=Q";       //!= 运算符
}
```

执行程序，输出结果：

```
P!=Q
```

8.6 函数调用运算符()

对一个类型,可以定义**函数调用运算符 operator**()。例如:

```
class Point {
    double x{}, y{};
public:
    double operator()(int n = 2) {
        if (n <= 1) return std::abs(x) + std::abs(y);
        double xn(x), yn(y);
        for (auto i = 1;i != n; i++) { xn *= x; yn *= y; }
        return xn + yn;
    }
    …
};
```

Point 类定义了一个**函数调用运算符 operator**()函数,其包含一个参数 n,该函数根据这个参数 n 计算 x^n+y^n。

对于一个 Point 对象 P,可以在这个对象后面传递**函数调用运算符**函数需要的参数 n,执行相应的计算。例如:

```
Point P{ 2,3 },
std::cout << P(2)<<'\t'<< P(3)<<'\n';      //P(2)和 P(3)调用了函数调用运算符 operator()
```

分别输出了 P 的 x^2+y^2 和 x^3+y^3 的值。P(2)这种对象名后面用圆括号传递实际参数的方式类似于函数调用。实际上,P(2)调用的是 P.**operator**()(2),即调用的是函数调用运算符 operator()函数。

P(2)这种使用方式使对象 P 看起来像一个函数一样,可以通过圆括号()接收参数。因此,将这个 Point 类的对象称为**函数对象**。即定义了函数调用运算符的类对象称为函数对象。

8.7 类型转换运算符

前面说过,带一个参数的构造函数实际上定义了一个从**参数类型**到**类类型**的隐式类型转换。反过来,也可以定义一个**类型转换运算符**用于将**类类型**隐式转换为其他的类型。**类型转换运算符**同样必须作为成员函数实现,其格式是:

```
operator type () const
```

即将类类型转换为 type 类型。例如,下面定义了 operator double() const 的类型转换运算符函数。这个函数可以自动将 Point 对象转换为 double 类型的值。

```
class Point {
    double x{}, y{};
public:
    operator double() const{
        return x * x + y * y;
    }
    Point(double x, double y) :x{ x }, y{ y }{}
};
```

下面的程序将 Point 对象 P 先隐式转换为 double 类型的值，再对 d 初始化：

```
int main() {
    Point P(3, 4);
    double d = P;                    //对象 P 隐式转换为 double 类型的值,再对 d 初始化
    std::cout << d << '\n';
}
```

一个参数的构造函数和类型转换运算符定义的隐式类型转换有时会引歧义。例如：

```
class A {
    //…
public:
    A(int);          //一个参数的构造函数定义了 int 类型到 A 类的自动类型转换
    operator int(); //类型转换运算符可以将 A 类型对象自动转换为 int 类型
    friend A operator + (const A& a1, const A& a2);   //2 个 A 类型的对象相加
};
```

上述代码带一个参数的构造函数 A(int)定义了一个从 int 到 A 的类型转换，而类型转换运算符 operator int()又可以将一个 A 对象转换为 int 类型值。对于下面的代码：

```
int main() {
    A a{1};
    int i = 1, z;
    z = a + i;      //错: 到底是 A + A 还是 int + int
}
```

main()函数的 a+i 到底是将 a 转换为 int 类型，然后是 2 个 int 类型值的相加，还是将 i 转换为 A 类型对象，然后 2 个 A 类型的对象相加？对于这种歧义的情况，就需要用显式类型转换，强制转换成需要的类型。如：

```
z = static_cast<int>(a) + i;  //或 z = (int)a + i;
```

或者

```
z = a + static_cast<A>(i);  //或 z = a + A(i);
```

8.8 自增和自减运算符

自增运算符(++)和自减运算符(——)都是一元运算符,且必须作为类的成员函数实现。自增和自减运算符还区分前缀或后缀。例如:

```
int x;
++x;
x++;
```

即对一个变量(对象),自增(自减)运算符可以位于该对象的左边或右边,分别称为**前缀自增**(自减)和**后缀自增**(自减)。因此,在重载这个自增(或自减)运算符时,需要重载实现前缀和后缀 2 个版本的自增(或自减)运算符函数。C++规定通过给运算符后面的圆括号里添加一个 int 型参数来区分是前缀还是后缀,尽管这个参数不会被使用。

例如:

```
class Point {
    double x{}, y{};
public:
    Point(double x, double y) :x{ x }, y{ y }{}
    Point& operator++();           //前缀自增运算符必须返回这个对象自身的引用
    const Point operator++(int);   //后缀自增运算符必须带一个 int 类型参数
                                   //且不能返回对象自身的引用
    …
};

Point& Point::operator++() {       //前缀++
    ++x; ++y;
    return * this;
}
const Point Point::operator++(int) {  //后缀++
    Point P( * this);              //暂时保存原来的对象值
    ++( * this);                   //对象自身自增
    return P;                      //返回的原来的对象值,一个临时变量
}
```

因为前缀自增返回的是对象自身,所以对它可以继续用++运算,而后缀++返回的是一个临时值,不能连续使用后缀++。如:

```
int main() {
    Point P{ 2,3 };
    ++ ++P;       //ok: 因为++P 返回的就是 P 自己,可以继续对它再用++运算
    (++P)++;      //ok:理由同上
    P++ ++;       //错: 因为 P++返回的不是自身引用,不能继续对 P++再用++运算
    ++(P++);      //错: 理由同上
}
```

8.9 可以重载的运算符

可以重载的运算符有：

```
+   -    *   /   %   ^   &   |   ~   !
+=   -=   *=   /=   %=   ^=   &=   |=   <<= >>=
> <  >=  <=   ==  !=  <<  >>
&&  ||  ++   --   ->   ->*   ,   *   T
[]  ()  new new[] delete delete[]
```

注意：

（1）这里的 T 表示的是类型转换运算符。

（2）有的运算符可能有 2 种含义，如 & 既可以作为位运算符也可以作为取地址符，—既可以作为减法运算符也可以作为负号运算符。

（3）有的运算符只能作为成员函数重载，如＝、[]、()、T(类型转换运算符)。

（4）有的运算符只能作为外部函数重载，如 new、new[]、delete、delete[]。

（5）有一些运算符不能被重载，如：

- ::，作用域运算符。
- .，成员访问运算符。
- .*，成员选择运算符。
- ?:，条件运算符。
- sizeof，查询对象的大小。
- typeid，查询对象的类型。

最后需要说明的是，运算符定义不能违背约定的语法，如不能将一元定义成二元或三元，反过来也是一样。也不应该违背运算符的语义，如不能将＋运算符定义成相乘的含义，赋值运算符的返回类型应返回引用而不是值。

8.10 实战：矩阵

矩阵是线性代数中的最基本概念，矩阵是一个二维数组，可以通过下标访问其中的某个元素。矩阵可以有加、减、乘等算术运算，当然也包含转置、求逆矩阵等运算。

由于计算机内存是一个一维结构，为了表示这种二维结构的矩阵，需要约定二维矩阵的元素在一维内存中的存放规则，假如采用按行存储的方式，即从第 1 行到第 2 行这种一行一行的方式依次将每行的元素放到一个一维的内存中。这个一维的内存可以用动态内存分配，根据矩阵的行列数计算出需要多大的内存空间(假设矩阵的行列数为 m 和 n)，可申请可存储 $m*n$ 个 double 的内存。

```
data = new double[m*n];
```

假设矩阵的行列下标 i、j 都是从 0 开始的，则下标(i,j)对应的元素在 data 数组中的下

标 $k=i*n+j$。因为下标 i 表示该元素的上面有 i 行，而在它所在这个行，前面又有 j 个元素，因此，下标为 (i,j) 的元素前面有 $i*n+j$ 个元素。

因为下标运算符[]只能带一个参数，因此，这里用可以带多个参数的函数调用运算符()来根据下标存取相应的矩阵元素：

```
class Matrix {
    double * data{nullptr};
    int m{}, n{};  //行数和列数
public:
    Matrix(const int m = 0, const int n = 0);
    explicit Matrix(const int m) :Matrix(m, m) {}
                //为防止将一个 int 类型值隐式自动转换为 Matrix 类型
                //这里用 explicit
    double operator() (const int i, const int j)const;
    double& operator()(const int i, const int j);
};

Matrix::Matrix(const int m, const int n) :m(m),n(n){
    if (m <= 0 || n <= 0) return;
    data = new double[m * n];
    if (!data) {  this->m = this->n = 0; return;  }
}
double Matrix::operator() (const int i, const int j)const {
    int k = i * n + j;
    return data[k];
}
double& Matrix::operator() (const int i, const int j) {
    int k = i * n + j;
    return data[k];
}
```

然后，可以写一段简单代码测试 Matrix。

```
int main() {
    Matrix A(3, 4);
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 4; j++)
            A(i, j) = i * 4 + j;     //A(i,j)调用的是引用版本的 operator()

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 4; j++)
            std::cout << A(i, j) << "  ";  //A(i,j)调用的是 const 版本的 operator()
        std::cout << std::endl;
    }
}
```

执行程序,输出结果:

```
0  1  2  3
4  5  6  7
8  9  10  11
```

Matrix 还应该能够复制、赋值,进行+、−、* 包括+=、−*、*=等运算。下面的代码将+=、−*、*=运算符作为成员函数实现,而+、−、*运算符作为友元函数实现:

```
class Matrix {
    double * data{nullptr};
    int m{0}, n{0};                 //行数和列数
public:
    Matrix(const int m = 0, const int n = 0);
    explicit Matrix(const int m) :Matrix(m, m) {}
                        //为防止将一个 int 类型值隐式自动转换为 Matrix 类型
                        //这里用 explicit
    double operator() (const int i, const int j)const;
    double& operator()(const int i, const int j);

    Matrix(const Matrix& M);        //拷贝构造函数
    Matrix& operator = (const Matrix& M);//赋值运算符函数

    Matrix & operator += (const Matrix &M);
    Matrix & operator -= (const Matrix &M);
    Matrix & operator *= (const Matrix &M);

    friend Matrix operator + (const Matrix &A, const Matrix &B);
    friend Matrix operator - (const Matrix &A, const Matrix &B);
    friend Matrix operator * (const Matrix &A, const Matrix &B);

    int rows()const { return m; }
    int cols() const{ return n; }
};

Matrix::Matrix(const int m, const int n) :m(m),n(n){
    if (m <=  0 || n <=  0) return;
    data  =  new double[m * n];
    if (!data) {this->m  =  this->n  =  0; return;       }
}
double Matrix::operator() (const int i, const int j)const {
    int k  =  i  *  n  +  j;
    return data[k];
}
double& Matrix::operator() (const int i, const int j) {
    int k  =  i  *  n  +  j;
    return data[k];
}
```

```
Matrix::Matrix(const Matrix& M) {
    int num = M.m * M.n;
    data = new double[num];              //申请一块内存
    if (data) {
      m = M.m; n = M.n;
      for (auto i = 0; i != num; i++)
          data[i] = M.data[i];
    }
}
Matrix& Matrix::operator = (const Matrix& M) {
    int num = M.m * M.n;
    double * temp = new double[num];    //申请一块新内存
    if (temp) {                          //数据赋值(复制)
        delete[] data;                   //释放原来的内存
        data = temp;                     //data 指向新内存块
        m = M.m; n = M.n;
        for (auto i = 0; i != num; i++)
            data[i] = M.data[i];
    }
    return * this;
}

Matrix & Matrix::operator += (const Matrix &M) {
    if (m != M.m || n != M.n) return * this;
    int num = m * n;
    for (auto i = 0; i != num; i++)data[i] += M.data[i];
    return * this;
}
Matrix & Matrix::operator -= (const Matrix &M) {
    if (m != M.m || n != M.n) return * this;
    int num = m * n;
    for (auto i = 0; i != num; i++)data[i] -= M.data[i];
    return * this;
}
Matrix & Matrix::operator *= (const Matrix &M) {
    //...补充你的代码
    return * this;
}

Matrix operator + (const Matrix &A, const Matrix &B) {
    Matrix C(A);       C += B;       return C;
}
Matrix operator - (const Matrix &A, const Matrix &B) {
    Matrix C(A);       C -= B;       return C;
}
Matrix operator * (const Matrix &A, const Matrix &B) {
    Matrix C(A);       C *= B;       return C;
}
```

编写一段简单代码测试一下刚添加的函数是否正确。

```
int main() {
    Matrix A(3, 4),B(3,4);

    for (int i = 0; i < A.rows(); i++)
        for (int j = 0; j < A.cols(); j++) {
            A(i, j) = i * A.cols() + j;
            B(i, j) = i * A.cols() + j;
        }

    for (int i = 0; i < A.rows(); i++) {
        for (int j = 0; j < A.cols(); j++)
            std::cout << A(i, j) << "  ";
        std::cout << std::endl;
    }
    std::cout << std::endl;

    Matrix C;
    C = A + B;
    for (int i = 0; i < C.rows(); i++) {
        for (int j = 0; j < C.cols(); j++)
            std::cout << C(i, j) << "  ";
        std::cout << std::endl;
    }
}
```

8.11 习题

1. 如何确定下列运算符是否应该作为类的成员函数?

(a) %　(b) %=　(c) ++　(d) ->　(e) <<　(f) &&　(g) ==

2. 为第 7 章的字符串类 String 添加下标运算符 operator[](替换原来的 get_ch 和 set_ch()功能)和输出运算符 operator <<功能。

3. 为下面的类 X 添加自增、自减、比较运算符的功能。

```
class X {
    int x;
public:
    X(x) :x{ x } {}
    int get_x() { return x; }
};
```

4. 为什么应该调用 operator+=()来实现 operator+()?

5. 根据自己对矩阵的理解,丰富完善矩阵类 Matrix。

6. 实现一个表示三维数学向量的类 Vector3,尽量用运算符重载定义对 Vector3 对象

进行运算。

7. 实现一个表示复数的类 complex，尽量用运算符重载定义对 complex 对象进行运算。至少应该实现加、减、乘、除、共轭和输入输出运算符。设 $z_1=a+b_i$、$z_2=c+d_i$ 是任意两个复数，s 是一个实数，这些运算规则如下。

加法：$z_1+z_2=(a+c)+(b+d)\mathrm{i}$

减法：$z_1-z_2=(a-c)+(b-d)\mathrm{i}$

乘法：$z_1*z_2=(ac-bd)+(bc+ad)\mathrm{i}$

数乘：$s*\ z_1=\ s*a+s*b\mathrm{i}$

除法：$z_1/z_2=(ac+bd)/(c^2+d^2)\ +((bc-ad)/(c^2+d^2))\mathrm{i}$

共轭：$\sim z_1=a-b\mathrm{i}$



派　生　类

9.1 继承与派生

9.1.1　继承关系和派生类

C++中类 class 用来描述一个概念，一个程序中通常有多个概念，这些概念之间可能存在一定的关系，如一个 Dog(狗)是一种特殊的 Animal(动物)，一个 Dog 具有 Animal 的所有属性，但还有 Dog 特有的属性，如狗喜欢刨骨头或啃骨头。如果在程序中分别用 2 个类来描述 Animal 和 Dog，则：

```
class Animal
{
    //…
};
class Dog
{
    //…
};
```

这是 2 个独立的类，尽管人们知道 Dog 是一种特殊的 Animal(或者说 Dog 也是一个 Animal)，但是编译器并不知道这 2 个类具有某种联系。

C++通过允许定义所谓的**派生类**这一语言特征，来明确地告知编译器 2 个类之间的**继承关系**，即可以从 Animal 类中定义一个派生类 Dog，让 Dog 在继承 Animal 类的属性基础上，再定义自己特有的属性。通过这种定义派生类来表达类(概念)之间的继承关系，使得概念的层次关系可以明确在程序中表示出来。代码如下所示：

```
class Animal
{
```

```
    //...
};
class Dog: public Animal
{
    //...
};
```

在Dog类的后面用冒号:跟上它所继承的类的类名Animal,就表示Dog类自动继承了Animal的所有属性。其中的关键字**public**是控制Animal的属性在Dog类中的可见性(后面会讲到)。

由于Dog类继承了Animal,因此就不需要在其中重复编写所有Animal共有的属性(包括成员变量和成员函数)的代码,而只需要关注Dog特有的属性,从而可以复用Animal的代码,提高了编程效率和程序的可靠性。因为Animal共有的属性只在Animal中编写,而不需要在它的派生类重复编写,减少了出现错误和不一致的概率。

9.1.2 is a 和 belong to

Dog(狗)是一种(**is a**)Animal(动物)表达了概念之间的继承关系,也称为**is a**关系。如公司中的经理也是一个雇员,即"经理**is a**雇员",和前面的Pong游戏中的"球(ball) **is a**精灵(sprite)"类似。

前面说过,除了这种**is a**关系外,概念之间也存在一种包含关系,称为**belong to**关系。如一个汽车中包含一个引擎或者说引擎是包含于或附属于(belong to)汽车的,一个员工是包含于或附属于(belong to)一个公司的。

包含于或附属于(belong to)关系是用类的成员变量表示的。如一个日期(date)包含了年(year)、月(month)、日(day),将年(year)、月(month)、日(day)定义为Date类对象的成员变量就表示了这种belong to关系。

9.1.3 派生类的定义

派生类的定义格式为:

```
class 派生类名: public 基类名
{
    派生类成员(数据成员和成员函数)
};
```

假设有一个类Sprite表示游戏中的精灵。

```
#include <iostream>
using std::cout;
class Sprite {
    double pos[2]{}, vel[2]{1.,1.};     //位置 pos 和速度 vel
public:
    Sprite(double *p = 0,double * v = 0) {
        if (p) { pos[0] = p[0]; pos[1] = p[1]; }
```

```
        if (v) { vel[0] = v[0]; vel[1] = v[1]; }
    }
    void update() { pos[0] += vel[0]; pos[1] += vel[1];    }    //根据速度更新位置
    void draw() {cout << "在("<< pos[0] << ','<< pos[1] << ")位置绘制精灵\n";
    }
};
```

该类定义了表示精灵位置和速度的 pos 和 vel 两个数据成员，除构造函数外，还定义了 update()函数用当前速度更新精灵的位置，用 draw()在游戏画面上绘制精灵，这里为了简单起见（不涉及图形输出），draw()用 std::cout 输出的方式模拟在游戏画面上绘制的过程。

Pong 游戏中的球可以用一个从 Sprite 类派生的类 Ball 表示如下。

```
class Ball: public Sprite {
    double radius{1.};
};
```

因为 Ball 是从 Sprite 派生的，因此 Ball 类的对象就自动继承了 Sprite 类对象的属性，如数据成员 pos 和 vel，还有成员函数 update()、draw()。Ball 类还定义自身的特殊成员变量 radius 表示球的半径。图 9-1 是 Sprite 类对象和 Ball 类对象的内存布局示意图。

执行下面的主函数：

```
int main() {
    Ball ball;
    ball.update();
    ball.draw();
}
```

Sprite对象		Ball对象	
pos	0	pos	0
	0		0
vec	1	vec	1
	1		1
		radius	

图 9-1　Sprite 类对象和 Ball 类对象的内存布局示意图

输出结果：

```
在(1,1)位置绘制精灵
```

9.1.4　成员的隐藏

上述的 ball.draw()调用的是基类的 draw()函数，这不符合要求，Ball 应该绘制的是自己，为此，可以在 Ball 中重新定义 draw()函数如下：

```
void draw() {
    cout << "绘制半径" << radius << ",圆心在("
        << pos[0] << ','<< pos[1] << ")的圆\n";
}
```

在这个 draw()函数中还访问了从基类继承下来的 pos 成员变量，但是 pos 成员变量是 Sprite 类私有的(private)变量，外界（包括 Sprite 的派生类 Ball）不能访问它。为了能在派生类中访问它，可以在 Sprite 类中用 protected 关键字将它们定义为 protected（保护的）成员。

```
protected:
    double pos[2]{}, vel[2]{ 1.,1. };//位置 pos 和速度 vel
```

现在，这些成员变量在 Ball 类也就能够访问了（即可见了），并且因为 Ball 是从 Sprite 通过 public 方式派生出来的，这些成员变量在 Ball 中也是 protected（保护）的。

Ball 类重新定义了 draw()成员函数，该函数就自动地隐藏了基类继承下来的 draw()，也就是说，通过 Ball 对象执行 draw()函数调用的是 Ball 类自身的这个 draw()函数，执行前面的 main()函数，输出结果是：

```
绘制半径 2,圆心在(1,1)的圆
```

如果想在派生类中访问被隐藏的基类的 draw()函数，需要用基类**作用域限定符**（如 **Sprite::**）。例如，修改派生类的 draw()函数：

```
void draw() {
    Sprite::draw();                  //调用基类 Sprite 的 draw()方法
    cout << "绘制半径" << radius << ",圆心在("
        << pos[0] << ','<< pos[1] << ")的圆\n";
}
```

执行前面的 main()函数，输出结果：

```
在(1,1)位置绘制精灵
绘制半径 2,圆心在(1,1)的圆
```

只要派生类的成员函数名和基类的成员函数名相同，在派生类中这个同名函数就隐藏了基类的同名函数，即使这 2 个函数的参数列表不同（即函数的签名不同）。

细心的读者会问：派生类能定义和基类的同名变量，以便隐藏基类的同名变量吗？答案是肯定的。

```
#include <iostream>
using std::cout;
class Base {
protected:
    int value{0};                    //定义了一个叫作 value 的 int 类型变量
public:
    Base(int v = 0) :value(v) {}
    void print() { cout << value << '\n'; }
};
class Derived :public Base {
    double value{ 1.5};              //定义了一个叫作 value 的 double 类型变量
public:
    void print(bool base) {
        if(base)
```

```
                cout << Base::value << '\t' << value << '\n';
            else
                cout << value << '\n';
        }
};
```

派生类 Derived 中的成员 value 和 print()都隐藏了基类的同名成员，且变量的类型或函数的签名都是不同的。执行下列主函数：

```
int main() {
    Base b;
    Derived d;
    b.print();
    d.print(true);
    //d.print();                    //编译错误：没有匹配的 Derived::print()
}
```

输出结果：

```
0
0       1.5
```

如果程序中直接使用 d. print()，则会出现编译错误：没有匹配的 Derived∷print()。这是因为 Derived 隐藏了基类 Base 的 print()，而调用 Derived 的 print(bool base)时必须传递一个参数。

派生类同名变量和成员函数会隐藏(hide)基类的同名变量和成员函数，即使函数签名是不一样的。

隐藏(hide)不同于**函数重载**(overloading)。函数重载是指同一个作用域中的不同签名函数，例如同一个类中的多个同名但不同签名的构造函数的重载。而派生类和基类有各自的作用域。

9.1.5 继承方式

一个类通过 private、protected、public 关键字修饰成员，控制成员对外界的可见性。关键字 private、protected、public 也可以用于定义派生类从基类的继承方式，即控制基类成员在派生类的可见性。

用 class 定义一个派生类时，如果没有指明派生方式，则默认是 private 继承方式。如：

```
class B {
    //...
};
class D :B {
    //...
};
```

即 D 是私有继承 B。相当于：

```
class D : private B {
    //...
};
```

用 struct 定义一个派生类时，如果没有指明派生方式，则默认是 public 继承方式。基类成员的对外可见性(private、protected、public)和派生类从基类的继承方式(private、protected、public)可产生 9 种组合，决定了基类成员在派生类中的可见性和对外的可访问性(private、protected、public)。表 9-1 是基类成员在派生类中的可见性和对外的可访问性。

表 9-1　基类成员在派生类中的可见性和对外的可访问性

private	protected	public
不管采用哪种继承方式，在派生类中都是不可访问的	如果是 private 继承方式，则在派生类中是 private 成员	如果是 private 继承方式，则在派生类中是 private 成员
	如果是 protected 继承方式，则在派生类中是 protected 成员	如果是 protected 继承方式，则在派生类中是 protected 成员
	如果是 public 继承方式，则在派生类中是 protected 成员	如果是 public 继承方式，则在派生类中是 public 成员

即基类的 private 成员在派生类中总是不可以访问的。如果采用的是 public 继承方式，则基类的 protected 和 public 成员在派生类中也是 protected 和 public 成员。如果采用的是 protected 继承方式，则基类的 protected 和 public 成员在派生类中都是 protected 成员。而如果采用的是 private 继承方式，则基类的 protected 和 public 成员在派生类中都是 private 成员。

如前所述，类的 public 成员可以被外界访问，而 protected 成员只能被该类及其派生类的成员函数或友元访问，private 成员只能被该类自己的成员函数或友元访问。

不管基类的成员在派生类中是否可见，派生类对象中总是存在这些基类成员变量的，只不过没法访问不可见的成员变量而已，是否存在和可见性无关。

9.1.6　基类指针和派生类指针

既然派生类对象也是一种特殊的基类对象，因此，可以将一个派生类对象当成一个基类对象使用。例如：

```
#include <iostream>
class B {
    int b{0};
public:
    B(int b = 0) :b(b) {}
    void print() { std::cout <<"B:"<< b; }
};
class D:public B {
```

```
    double d{ 2.5 };
public:
    D(double d = 0) :d(d) {}
    void print() {
        std::cout << "D:";
        B::print(); std::cout <<'\t'<< d ;
    }
};

int main() {
    B b{ 1 };
    D d{3.14};
    b.print();  std::cout << '\n';
    d.print();  std::cout << '\n';
    b = d;
    b.print();  std::cout << '\n';
}
```

派生类对象可自动转换为基类类型，如执行 b = d，将 D 的对象 d 赋值给 B 的对象 b 时，d 被隐式转换为基类类型 B 的对象，然后再赋值给 b。但派生类对象赋值给基类对象时，会产生切割，即被赋值的基类对象只有基类部分的成员变量，派生类部分的成员变量（如 D 的 d 成员变量）丢失了，最后的 print()输出了基类成员变量的值。

执行程序的结果是：

```
B:1
D:B:0  3.14
B:0
```

避免"对象被切割"的更好的方法是用基类指针指向派生类对象，或者说将派生类指针赋值给基类指针变量。如：

```
int main() {
    B b{ 1 };
    D d{ 3.14 };
    B *p = &b;                          //基类指针 p 指向基类对象 b
    p->print(); std::cout << '\n';
    p = &d;                             //基类指针 p 指向派生类对象 d
    p->print(); std::cout << '\n';
}
```

即基类指针也可以存储派生类对象的地址（p = &d）。然而，因为 p 是基类 B 的指针变量，不管 p 指向的是基类 B 还是派生类 D 的对象，p->print()调用的都是基类 B 的 print()函数，输出的总是其指向对象的基类成员：

```
B:1
B:0
```

这个问题将在9.4节中解决。

不管怎么说，派生类指针可以自动隐式转换为基类指针类型，但反过来则不行：

```
D *q = p;                         //错：基类指针不能自动隐式转换为派生类指针类型
```

运行程序，编译器会报告错误：

```
error C2440: '初始化':不能将'B * '转换为'D * '
```

但可以通过强制类型转换将基类指针类型转换为派生类指针类型：

```
D *q = static_cast<D*>(p);        //强制将B*转换为D*
q = static_cast<D*>(&b);          //强制将B*转换为D*
```

运行程序，编译器都能通过，不会报告任何错误。对于后一个强制类型转换，使得D*类型指针q指向的实际对象是一个B类对象，如果通过这个q去访问派生类的成员会导致严重错误。但如果q指向的实际对象就是D类的对象，则没有问题。例如：

```
D *q = static_cast<D*>(p);      //基类指针p指向的是D类对象,因此q指向的是D类对象
q->print(); std::cout << '\n';  //q指向的是D类对象,调用D的print()当然没问题
```

q虽然是从B*类型指针p强制转换的，但它实际指向的是一个D类对象，通过q调用D的print()函数当然没有任何问题，输出如下：

```
D:B:0  3.14
```

强制类型转换应该尽量避免，即使必须用强制类型转换，也要小心使用。

9.2 派生类的构造函数和析构函数

在基类和派生类的构造函数中添加一些输出语句，如：

```
#include <iostream>
using std::cout;
class B {
    int b{ 0 };
public:
    B() { cout << "B类构造函数\n"; }
};
class D :public B {
    double d{ 2.5 };
public:
    D() { cout << "D类构造函数\n"; }
};
```

然后在 main()函数中定义一个 D 类的变量 d：

```
int main() {
    D d;
}
```

执行程序，输出结果：

```
B类构造函数
D类构造函数
```

即在定义(创建)D 类对象时，D 类的构造函数实际上先调用了基类 B 的构造函数对这个对象的基类(B)部分(即 B 的 b 成员变量)进行初始化，然后才是对派生类自身的数据成员进行初始化。

假如 B 类派生自其他类，如 A，即 A 是 D 的间接基类，那么在创建 D 类对象时，先执行的是 B 的基类即 A 的构造函数，然后才执行 B 的构造函数，最后才是 D 的构造函数，即从最顶端的基类开始依次执行派生层次上的各个基类的构造函数，最后才是派生类自己的构造函数。

同样地，在销毁一个对象时，却是反其道而行，即先执行派生类自己的析构函数，然后是其上层的基类的析构函数，直到最上层基类的析构函数。可以为上面的类添加析构函数来验证这一点。

```
class B {
    int b{ 0 };
public:
    B() { cout << "B类构造函数\n"; }
    ~B() { cout << "B类析构函数\n"; }

};
class D :public B {
    double d{ 2.5 };
public:
    D() { cout << "D类构造函数\n"; }
    ~D() { cout << "D类析构函数\n"; }
};
```

再执行上面的程序，输出结果：

```
B类构造函数
D类构造函数
D类析构函数
B类析构函数
```

即 main()函数执行完，销毁 D 类对象 d 时，先执行的是 D 的析构函数，然后才是其基类 B 的析构函数，因此，先输出“D 类析构函数”，后输出“B 类析构函数”。

派生类的构造函数对对象的基类部分初始化时，调用的是基类默认的构造函数，也可以

在派生类的初始化成员列表中调用基类的其他构造函数。

```
class B {
      int b{ 0 };
public:
    B() { cout << "B类默认构造函数\n"; }
    B(int b):b(b) { cout << "B类构造函数\n"; }
    ~B() { cout << "B类析构函数\n"; }
};
class D :public B {
    double d{ 2.5 };
public:
    D():B(2) { cout << "D类默认构造函数\n"; }    //在初始化成员列表中调用基类构造函数
    ~D() { cout << "D类析构函数\n"; }
};
```

D的默认构造函数调用基类的非默认构造函数B(int b)对基类部分进行构造。再执行上面的main()程序，输出结果：

```
B类构造函数
D类默认构造函数
D类析构函数
B类析构函数
```

下面的代码中，派生类D的构造函数在其函数体内调用基类的构造函数：

```
D() { B(2); cout << "D类默认构造函数\n"; }
```

其中，B(2)只是在D的构造函数内部创建了一个局部变量，而并不是对要创建的D类对象的基类部分进行初始化，当D的构造函数执行完，这个局部变量就被销毁了。因此，D的默认构造函数此时仍然调用B的默认构造函数对其基类部分进行构造。如果采用这个构造函数，执行上述main()函数，输出结果：

```
B类默认构造函数
B类构造函数
B类析构函数
D类默认构造函数
D类析构函数
B类析构函数
```

因此，**只能在派生类的初始化成员列表中调用基类的构造函数对要创建的派生类对象的基类部分进行初始化**。

因为派生类的构造函数要调用基类构造函数对派生类对象的基类部分初始化，如果基类没有默认构造函数，则派生类的构造函数函数**必须在初始化成员列表中显式调用基类构造函数并提供必须的参数。因此，基类构造函数需要的参数通常应该在派生类的构造函数的形参列表中出现**。

```
#include <iostream>
#include <string>
using std::cout;
using std::string;
class B {
    int b{ 0 };
    string name{};
public:
    B(int b,string n):b(b),name(n) { cout << "B类构造函数\n"; }
    ~B() { cout << "B类析构函数\n"; }

};
class D :public B {
    double d{ 2.5 };
public:
    D() { cout << "D类默认构造函数\n"; }
    ~D() { cout << "D类析构函数\n"; }
};
int main() {
    D d;
}
```

上述程序将产生编译错误："error C2512："B":没有合适的默认构造函数可用"。这是因为B没有默认的构造函数,派生类的构造函数D()无法调用基类构造函数。正确的做法是在D的构造函数的初始化成员列表中调用基类B的某个构造函数并提供必须的参数：

```
D(double d,int b,string n):B(b,n),d(d) { cout << "D类构造函数\n"; }
```

上述代码中,D的构造函数的形参列表中包含了可以调用B的构造函数的参数(即int型参数b和string型参数n),并在初始化成员列表中调用了基类B的构造函数B(b,n)。

当然,main()函数中定义D类型的对象d时需要传递相应的参数才行：

```
int main() {
    D d(3.0,2,"helo");
}
```

执行程序,输出结果：

```
B类构造函数
D类默认构造函数
D类析构函数
B类析构函数
```

当然,也可以在派生类的初始化参数列表中直接给基类构造函数提供确定的参数值：

```
D():B(2,"name"),d(d) { cout << "D类默认构造函数\n"; }
```

但这种硬编码的方式使得 D 的所有对象的基类部分都具有一样的内容。

下面介绍拷贝构造函数。

定义一个类对象并用同一个类的其他对象初始化时，会自动调用拷贝构造函数，如果自己没有定义拷贝构造函数，编译器会自动生成这个类的拷贝构造函数。

为了观察派生类的拷贝构造函数如何调用基类的拷贝构造函数，可以在基类和派生类的拷贝构造函数中添加输出语句，例如：

```
#include <iostream>
using std::cout;
class B {
public:
    B() { cout << "B类默认构造函数\n"; }
    B(const B& b) { cout << "B类拷贝构造函数\n"; }
};
class D :public B {
public:
    D() { cout << "D类默认构造函数\n"; }
    D(const D& d) { cout << "D类拷贝构造函数\n"; }
};
int main() {
    D d,d2(d);
}
```

执行程序，输出结果：

```
B类默认构造函数
D类默认构造函数
B类默认构造函数
D类拷贝构造函数
```

最后 2 行输出是因为执行了 d2(d)，即调用了 D 的拷贝构造函数。D 的拷贝构造函数先调用基类 B 的默认构造函数，然后才调用自己的拷贝构造函数。

即基类部分调用的是 B 的默认构造函数而不是拷贝构造函数，即“基类部分没有能够拷贝构造”。显然这不符合要求，特别是基类 B 包含资源时，会出现严重问题。

解决办法是在派生类拷贝构造函数的初始化成员列表中调用基类的拷贝构造函数。如：

```
#include <iostream>
using std::cout;
class B {
public:
    B() { cout << "B类默认构造函数\n"; }
    B(const B& b) { cout << "B类拷贝构造函数\n"; }
};
class D :public B {
public:
    D() { cout << "D类默认构造函数\n"; }
```

```
    D(const D& d) :B{d}              //在初始化成员列表里调用基类的拷贝构造函数
    {
        cout << "D类拷贝构造函数\n";
    }
};
int main() {
    D d, d2(d);
}
```

执行程序,输出结果:

```
B类默认构造函数
D类默认构造函数
B类拷贝构造函数
D类拷贝构造函数
```

可以看到,现在对基类部分执行的是基类的拷贝构造函数。

下面的代码假设B和D都有一些成员变量。派生类的拷贝构造函数的初始化成员列表中同样调用了基类的拷贝构造函数。

```
#include <iostream>
#include <string>
using std::cout;
using std::string;
class B {
    int b{ 0 };
    string name{};
public:
    B(const B& b):b(b.b),name(b.name) { cout << "B类拷贝构造函数\n"; }
    B(int b,string n):b(b),name(n) { cout << "B类构造函数\n"; }
};
class D :public B {
    double d{ 2.5 };
public:
    D(const D& d) :d(d.d),B(d) { cout << "D类拷贝构造函数\n"; }
    D(double d,int b,string n):B(b,n),d(d) { cout << "D类构造函数\n"; }

};
int main() {
    D d(3.0,2,"helo");
    std::cout << '\n';
    D d2(d);
    std::cout << '\n';
}
```

执行程序,输出结果:

```
B类构造函数
D类构造函数
```

```
B类拷贝构造函数
D类拷贝构造函数
```

9.3 多继承和虚基类

9.3.1 多继承

前面的派生类都只有一个直接基类，C++中还可以从多个直接基类定义一个派生类，这种继承方式称为**多继承（multiple inheritance）**。如子女直接从父亲和母亲那里继承父母双方的遗传特性，再如一个高校的学生可能会作为助教，即兼有了教师和学生的特征。

定义多继承的派生类的方式类似于单继承，只不过多个直接基类名之间需要用逗号隔开。假如一个类 D 直接继承了 A、B、C，其定义格式如下：

```
class D: public A, protected B, private C {
    //…D类的成员
};
```

即 D 分别以 public、protected、private 等不同继承方式继承了基类 A、B、C 的属性。再如：

```
#include <iostream>
class Shape {
public:
    void draw() {std::cout << "绘制一般形状\n"; }
};
class Color {
    int color{0};
public:
    int get_color() { return color; }
};
class Circle:public Shape, public Color {
public:
    void draw() { std::cout << "绘制圆\n"; }
};

int main() {
    Circle c;
    std::cout << c.get_color() << '\n';
    c.draw();
}
```

程序执行结果：

```
0
绘制圆
```

该程序中，表示圆的类 Circle 从代表形状和颜色的直接基类 Shape 和 Color 派生。Circle 类的 draw()覆盖了基类 Shape 的同名函数 draw()，因此，当通过 Circle 对象 c 调用 draw()时，调用的就是派生类 Circle 自己的 draw()函数，而 c.get_color()调用的是从基类 Color 继承下来的方法 get_color()。

如果想通过派生类对象调用基类 Shape 的 draw()，可以通过基类作用域限定，即用 **Shape::**，代码如下：

```
c.Shape::draw();
```

和单继承一样，如果某个直接基类没有默认构造函数，则派生类的构造函数中至少应该包含这个直接基类的构造函数的参数，且在派生类构造函数的初始化成员列表中调用这个基类的构造函数并提供必须的参数，以便对派生类对象的这个基类部分进行初始化。假如 Corlor 定义了如下的构造函数：

```
Color(int c) :color(c) {}
```

那么派生类 Circle 的构造函数中必须至少包含用于调用 Color 构造函数的参数，且在 Circle 的构造函数的初始化成员列表中调用 Color 的这个构造函数。因此，Circle 必须定义一个含 int 类型参数的构造函数：

```
Circle(int color) :Color(color) {}
```

当然，如果 Circle 构造函数不包含 int 类型参数，则其初始化参数列表中要调用 Color 的构造函数必须传递一个可转换为 int 类型的文字量。如：

```
Circle() :Color(3) {}
```

如果一个派生类的不同基类包含了同名的数据成员或同样签名的函数成员，当通过该派生类对象访问这个成员时，可能会产生二义性问题。如：

```
class USBDevice{
private:
    long m_id;
public:
    USBDevice(long id): m_id(id){}
    long getID() { return m_id; }
};

class NetworkDevice{
private:
    long m_id;
public:
    NetworkDevice(long id): m_id(id){}
    long getID() { return m_id; }
};
```

```
class WirelessAdapter : public USBDevice, public NetworkDevice{
public:
    WirelessAdapter(long usbId, long networkId)
        : USBDevice(usbId), NetworkDevice(networkId)     {      }
};
int main(){
    WirelessAdapter  wa(5442, 181742);
    std::cout << wa.getID();        //调用哪一个 getID()
    return 0;
}
```

该程序中用 USBDevice 和 NetworkDevice 分别表示 USB 设备和网络设备，这两个类中都有同名的变量 m_id 和方法 getID()。从它们直接派生的类 WirelessAdapter 描述无线网卡。通过类 WirelessAdapter 的对象 wa 调用 getID()(即 wa.getID())，到底调用的是哪个基类的 getID()？相信读者应该知道怎么解决这个二义性问题了。

9.3.2 虚基类

多继承时可能一个派生类对象中有多份间接基类对象，例如：

```
#include <iostream>
#include <string>
using namespace std;
class Person {                                  //人
protected:
    string name{"noname"};
};

class PartyMember:public Person{                //党员
protected:
    string party{ "RP" };
};

class Teacher :public Person{                   //教师
protected:
    string title{ "TA" };                       //职称
    string profession{ "CS" };                  //专业
};

class TeacherPM :public Teacher,PartyMember{    //教师党员
};
```

如图 9-2(a)所示，TeacherPM 对象将包含其直接基类 PartyMember 对象和 Teacher 对象，而 PartyMember 对象和 Teacher 对象都各自包含一个 Person 对象。可以用 sizeof()运算符检查每种类型对象占用的内存大小：

```
int main() {
    Person p;
    PartyMember pm;
    Teacher t;
```

```
    TeacherPM tpm;
    string s{"hello"};
    cout << sizeof(s)<<"\n"
    cout << sizeof(p) << '\t'<< sizeof(pm) << '\t'<< sizeof(t) << '\t'
        << sizeof(tpm) << '\n';
    return 0;
}
```

程序输出结果：

```
28
28        56        84        140
```

在作者计算机的Windwos系统和VS 2017环境下，一个string类对象如s默认占用28字节内存。Person类对象p只有一个string类成员变量，因此，占用28字节，PartyMember类对象pm包含从Person继承下来的name一共2个string成员变量，占用56字节。同样，Teacher类对象t一共3个string变量，共84字节。而TeacherPM对象tpm包含从2个直接基类继承下来的成员，一共5个string，占用140字节。

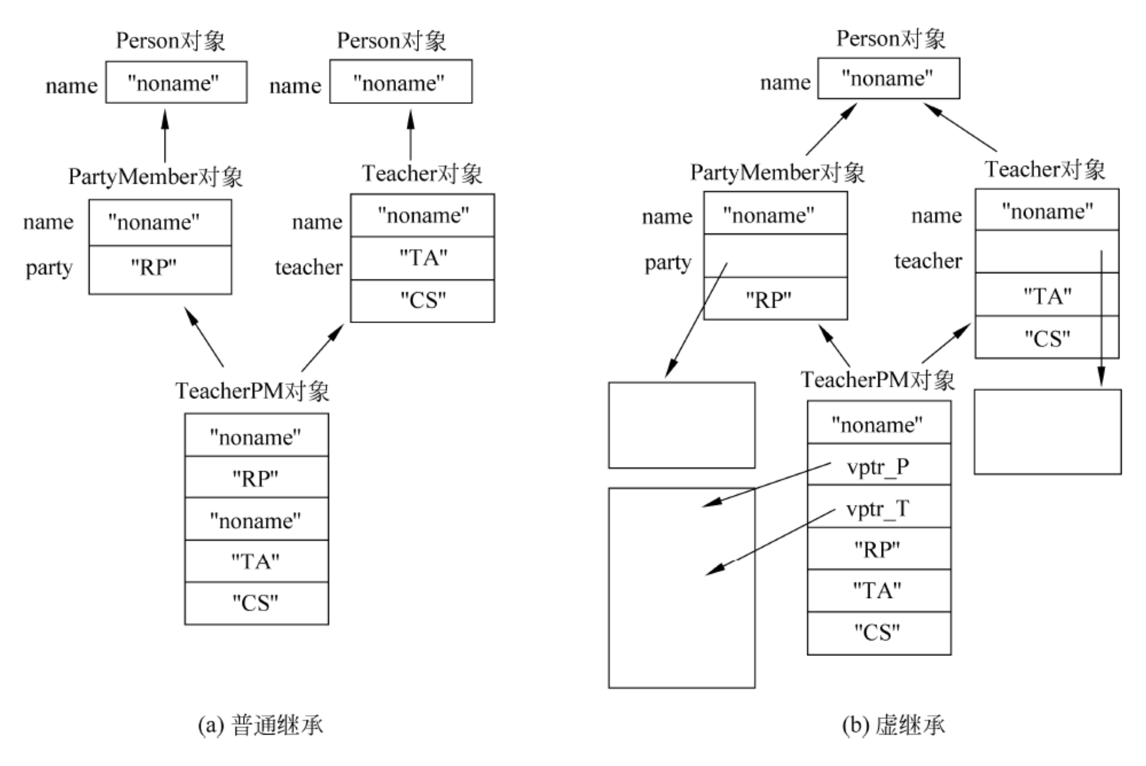


图 9-2 TeacherPM对象继承图及内存布局

TeacherPM对象tpm从2个直接基类PartyMember和Teacher都继承下来一份Person的数据，如name成员，而一个人不应该有重复的2个名字。为了避免这种间接基类对象在派生类中出现多个副本，可以在定义派生类时，声明继承的基类为虚基类，即修改代码如下：

```
class PartyMember : virtual Person{                    //党员
protected:
    string party{ "RP" };
```

```
};
class Teacher : virtual Person{                     //教师
protected:
    string title{ "RP" };                           //职称
    string profession{ "CS" };                      //专业
};
```

即 PartyMember 和 Teacher 以**虚继承**的方式继承了 Person 的数据，针对这种派生方式的基类 Person 被称为派生类的**虚基类**。派生类对象的内存中除包含其直接虚基类的成员变量外，还包含一个相应的指向虚函数表的指针，如图 9-2(b)所示，表示对应的虚基类的函数在虚函数表中的位置。现在如果一个类如 TeacherPM 同时从 PartyMember 和 Teacher 继承，其虚基类 Person 数据在派生类对象中只有一份而不会出现多份。

```
class TeacherPM :Teacher,PartyMember{               //教师党员
};
```

再执行上述的 main()函数，程序输出结果：

```
28
28      60      88      120
```

当然，由于 PartyMember 和 Teacher 都是从虚基类派生的，相应的对象的内存比原先的大 4 字节(即一个指针变量占用的内存)，从而 Teacher 对象的内存就变成 140＋4＋4－28 ＝ 120 字节。

9.4 多态

多态是面向对象编程的一个重要特性。在 C++ 中多态是指用一个基类指针(或引用)可以指向(或引用)不同类型的派生类对象，当通过这个指针(或引用)去调用一个称为“**虚函数**”的成员函数时，会根据指针指向的(或引用变量引用的)对象的实际类型而调用该类型的同名的虚函数。

例如，在一个画图程序中，有一个表示形状的类 Shape，从这个类 Shape 可以派生出不同的具体形状类，如 Circle(圆)、Line(线)、Rectangle(矩形)、Triangle(三角形)，从 Rectangle 又可以派生出 Square(正方形)。假如这些类都有一个叫作 draw()的绘制函数用于绘制自己，可以用一个 Shape * 类型的指针(该指针变量名假设叫作 p)去指向不同派生类型的对象，当通过这个指针去调用 draw()函数(p－> draw())时，如果能根据 p 指向的对象的实际类型去调用这个类型的 draw()函数，即根据指针(或引用变量)指向(或引用)的对象的实际类型去调用该类型的同名函数，这样一种特性就称为**多态(性)**。即根据指向或引用实际对象的类型不同产生不同的行为效果。

9.4.1 对象的切割和类型转换

前面说过，一个派生类对象中包含其基类对象部分，因此，一个派生类对象也是一个基

类对象，而反过来则不行。如一个 Dog(狗)也是一个 Animal(动物)，但不能说一个 Animal(动物)也是一个 Dog(狗)。因此，可以将一个派生类对象赋值给一个基类对象，此时会自动进行派生类型到基类类型的隐式类型转换，将派生类对象的基类对象部分赋值给基类对象，而丢弃掉派生类对象自己特有的属性。这种现象称为**切割**。尽管语法上是可行的，但切割后得到的基类对象丢失了原来的派生类对象的特有属性。

假设用 Person 和 Student 分别表示普通人和学生，则：

```
class Person {                                    //人
protected:
    string name{ "noname" };
public:
    Person(string n) :name(n) {}
    void print() { cout << name << '\t'; }
};
class Student:public Person {                     //学生
public:
    double score{0};
    Student(string n,double s) :Person(n) ,score(s){}
    void print() { Person::print(); cout << score << '\n'; }
};
```

可以将一个 Student 对象赋值给一个 Person 对象，会自动进行类型转换，但反过来不行：

```
int main() {
    Person p{ "Li Ping" };
    Student s{ "Zhang wei",60 };
    p = s;                        //派生类对象可以赋值给基类对象,但产生了切割
    cout << p.score;              //错: p 是 Person 对象,没有 score 属性
    s = p;                        //错: 不能将 Person 对象赋值给 Student 对象
}
```

执行“p=s;”后，p 是一个 Person 对象，执行语句“cout << p.score;”编译器将报告错误：

```
error C2039: "score": 不是"Person"的成员
```

这是因为 Person 对象没有 score 成员。

不能将一个基类对象赋值给一个派生类对象，如 s = p，试图将 Person 对象赋值给 Student 对象，编译器会报告错误：

```
error C2679: 二进制" = ": 没有找到接受"Person"类型的右操作数的运算符(或没有可接受的转换)
```

9.4.2　基类指针(引用)和向下类型转换

1. 基类指针(引用)指向(引用)派生类对象

尽管派生类对象可以当作基类对象使用，但直接将派生类对象赋值给基类对象将导致

派生类对象被切割。

既然派生类对象也是基类对象，那么也可以将派生类对象指针当作基类对象指针或将派生类对象的引用当作基类对象的引用。使用基类指针指向派生类对象或使用基类引用去引用派生类对象都不会导致派生类对象被切割。

例如：

```
int main() {
    Person p{ "Li Ping" }, *pp = &p;
    Student s{ "Zhang wei",60 };
    pp = &s;                        //派生类对象 s 的地址(指针)赋值给基类指针变量
    Person &r = s;                  //基类引用变量可以引用派生类对象
    pp->print();
    r.print();
}
```

可以用基类指针 pp 指向派生类对象，也可以用基类引用 r 引用派生类对象。程序运行结果：

```
Zhang wei
Zhang wei
```

尽管用基类指针(或引用)指向(或引用)了派生类对象，但通过它们调用的 print()函数仍然是基类而不是派生类的 print()函数。显然这是一个问题。解决这个问题的方法是将 print()定义为**虚函数**。

2. static_cast<>和向下类型转换

反过来，不能直接将基类指针(或引用)赋值给派生类指针(或引用)变量。下面的语句是错误的。

```
Student *ps = pp;
```

但可以通过强制类型转换 static_cast<>将一个基类指针(或引用)转换为一个派生类的指针(或引用)，称之为**向下类型转换**(downcasting)。static_cast<>试图在编译期间使用隐式转换和用户定义类型转换的组合在类型之间进行转换。如：

```
Student *ps = static_cast<Student *>(pp);     //将 Person * 强制转换为 Student *
ps->print();
ps = static_cast<Student *>(&p);              //将 Person * 强制转换为 Student *
ps->print();
```

pp 虽然是基类指针，但实际指向的是派生类对象，即存储的是派生类对象的地址，static_cast<Student *>(pp)将 pp 强制转换为派生类指针，不会造成任何问题。

但 p 是一个基类对象，将 &p 强制转换为派生类指针，虽然语法上没有问题，但后面的代码可能因为 ps 是一个派生类指针，而去访问派生类的特有成员，会造成严重的后果(如非法内存访问)。这段代码的输出是：

```
Zhang wei 60
Li Ping 2.27653e-305
```

可以看到，因为 ps 指向的是派生类对象，两次 ps->print()都是调用的派生类对象的 print()函数，第一个结果是正确的，因为 ps 指向的对象确实是派生类对象，而第二次结果就不正确，因为此时 ps 指向的实际是一个基类对象而不是派生类对象。

static_cast<>可以用于基本类型之间的强制类型转换，也可以用于具有继承和派生关系的指针(或引用)类型之间的类型转换。

将一个派生类指针(或引用)转换为一个基类的指针(或引用)，称为**向上类型转换**(upcasting)，这种转换可以自动隐式进行，不需要用 static_cast<>。

下面代码中既包含了用 static_cast<>的向下类型转换，也包含了隐式的向上类型转换。

```
#include <iostream>
struct B {
    int m = 0;
    void print() const {std::cout << "Hi, this is B!\n";}
};
struct D : B {
    void print() const {std::cout << "Hi, this is D!\n";}
};
class X {};
int main() {
    int n = static_cast<int>(3.14);       //基本类型之间的 static_cast
    D d;
    B& br = d;                            //向上类型转换：派生类引用可以自动隐式转换为
                                          //基类引用
                                          //也可以用 static_cast<>: B& br = static_cast
                                          //<B&>d;
    br.print();
    D& dr = static_cast<D&>(br);          //向下类型转换(downcast)
    dr.print();

    D* dp = new D;
    B* bp = dp;                           //向上类型转换：派生类指针可以自动隐式转换为
                                          //基类指针
                                          //也可以用 static_cast<>: B* bp = static_cast
                                          //<B*>d;
    D* dp2 = static_cast<D*>(bp);         //向下类型转换
    X* xp = static_cast<X*>(dp);          //编译错误：无法从"D *"转换为"X *"

    void *p = dp;                         //任何指针都可以转换为 void*
    D* dp3 = static_cast<D*>(p);          //将 void* 强制转换为 D*
    return 0;
}
```

static_cast<>只能用于相关类型，如具有继承和派生关系的指针(或引用)类型之间的

强制类型转换。2 个不同类型指针(或引用)如 D * 和 X * 之间是不能用 static_cast <>强制类型转换的,否则会出现编译错误:

```
error C2440: "static_cast": 无法从"D *"转换为"X *"
```

9.4.3 虚函数和多态

在一个类的成员函数声明前添加关键字 **virtual**,这个成员函数就变成了所谓的**虚函数**。所有从这个类直接或间接派生的派生类不管有没有定义这个函数,都具有了这个虚函数(假设访问控制和继承控制保证该函数是可见的话),这些派生类就具有了所谓的**多态性**。也就是说当通过基类指针(或引用)调用这个虚函数时,程序会根据指针(或引用)实际指向(或引用)的对象的实际类型去调用这个类型的这个虚函数。

例如,可以将前面的 Person 类的 print()函数定义为虚函数:

```
class Person {                          //人
protected:
    string name{ "noname" };
public:
    Person(string n) :name(n) {}
    virtual void print() { cout << name << '\t'; }
};
```

另外,只要在基类中用关键字 virtual 声明了虚函数,派生类中这个虚函数前加不加 virtual 关键字,这个函数都是虚函数。例如再定义一个派生类 Teacher:

```
class Teacher :public Person {          //学生
public:
    string title{"讲师"};               //职称
    Teacher(string n, string t) :Person(n), title(t) {}
    void print() { Person::print(); cout << score << '\n'; }
};
```

用下列 main()函数测试一下:

```
int main() {
    Person p{ "Li Ping" }, *pp = &p;    //pp 指向了 Person 对象
    Student s{ "Zhang wei",60 };
    Teacher t{ "王强","教授" };
    pp->print();                        //调用的是 Person 的 print()
    cout << '\n';
    pp = &s;                            //pp 指向了 Student 对象
    pp->print();                        //调用的是 Student 的 print()
    cout << '\n';
    pp = &t;                            //pp 指向了 Teacher 对象
    pp->print();                        //调用的是 Teacher 的 print()
    cout << '\n';
```

```
    Person &r = s;                    //基类引用变量可以引用派生类对象
    r.print();
}
```

程序运行结果：

```
Li Ping

Zhang wei
60

王强
教授

Zhang wei
60
```

可以看到，不管基类指针 pp 指向哪个(派生)类对象，都能调用这个对象所属类型的 print()虚函数。对于基类引用变量 r 也是如此。

dynamic_cast<>主要用于具有多态性的层次继承结构的类之间的指针(或引用)的向上、向下和侧向转换。它是在程序运行期间根据指针(或引用)指向(或引用)的对象的实际类型确定是否能安全地进行指针(或引用)类型的转换。其格式是：

```
dynamic_cast<Type* >(p)
dynamic_cast<Type& >(r)
```

即在运行时，将指针 p(或引用 r)转换为类型 Type*(或 Type&)。如果不能进行类型转换，对于指针，则返回空指针；对于引用，则抛出一个异常(错误)。

dynamic_cast<>主要用于将一个基类指针(或引用)转换为一个派生类的指针(或引用)，即**向下类型转换**(downcasting)。**向上类型转换**(upcasting)可以使用也可以不使用 dynamic_cast。

先看一个简单的例子：

```
#include <iostream>
using std::cout;
struct Base {
    virtual ~Base() {}
};

struct Derived: Base {
    virtual void name() {}
};
int main(){
    Base* b1 = new Base;
    if(Derived* d = dynamic_cast<Derived*>(b1)) {
        std::cout << "downcast from b1 to d successful\n";
        d->name();
```

```
    }

    Base * b2 = new Derived;
    if(Derived * d = dynamic_cast<Derived *>(b2)) {
        std::cout << "downcast from b2 to d successful\n";
        d->name();
    }

    delete b1;
    delete b2;
}
```

运行程序，输出结果：

```
downcast from b2 to d successful
```

可见，用 dynamic_cast 将一个基类指针（或引用）强制转换为派生类指针（或引用）时，只有基类指针（或引用）指向（或引用）的实际类型是这个派生类型对象时，转换才能成功。

再看一个复杂一点的例子。必须是多态的才能使用运行时检查的 dynamic_cast。

```
#include <iostream>
struct V {
    virtual void f() {};
};
struct A : virtual V {};
struct B : virtual V {
  B(V* v, A* a) {
    //构造过程中的类型转换(看下面 D 的构造函数的调用)
    dynamic_cast<B*>(v);   //没问题: v 的类型是 V*, 而 V 是 B 的基类,v 可以转换为 B* 类型
    dynamic_cast<B*>(a);   //不可预知: undefined behavior: a 的类型是 A*,但 A 不是 B 的基类
  }
};
struct D : A, B {
    D() : B(static_cast<A*>(this), this) { }
};
int main(){
    D d;
    A& ra = d;                  //向上类型转换:派生类引用转换为基类引用.可以使用也可以不使
                                //用 dynamic_cast
    D& rd = dynamic_cast<D&>(ra);      //向下类型转换:基类引用转换为派生类引用
    B& rb = dynamic_cast<B&>(ra);      //侧向类型转换:从 A& 转换为 B&
    A a;
    D& rda = dynamic_cast<D&>(a);      //运行时错误:因为实际对象 a 不是 D 类型
    B& rba = dynamic_cast<B&>(a);      //运行时错误:因为实际对象 a 不是 B 类型
}
```

该程序虽然可以编译通过，但在运行时会出错，因为最后 2 句中使用 dynamic_cast<> 时，实际对象是 a 而不是 D 或 B 类型，因此无法进行类型转换。

9.4.4 虚函数的一些语法规则

1. 类体外定义虚函数

和 inline 内联成员函数一样，类体外定义的虚函数不能有关键字 virtual，且必须在类体里的函数声明前添加关键字 virtual。如：

```
class Person {                                //人
protected:
    string name{ "noname" };
public:
    Person(string n) :name(n) {}
    virtual void print();                     //类体里的函数声明前加 virtual 关键字
};
void Person::print() { cout << name << '\n'; }  //类体外的函数定义前不能有 virtual 关键字
```

2. 虚函数的签名和返回类型

函数的签名是指函数名、形参列表和函数的修饰符，如 const。派生类和基类的虚函数的签名必须相同。此外，虚函数还要求虚函数的返回类型要么相同，要么是该类的指针或引用类型。例如：

```
#include <iostream>
class B {
public:
    virtual B& f() { return *this; }
    virtual int g() { std::cout << "g\n"; return 0; }
};
class D:public B {
public:
    D& f() { return *this; }
    double g() { std::cout << "g\n"; return 0.; }
};
```

编译时对虚函数 g()报告错误：

```
error C2555: "D::g": 重写虚函数返回类型有差异,且不是来自"B::g"的协变
```

因为这是 2 个同样签名(函数名和参数列表相同)的函数，作为虚函数其返回类型既不同也不是该类的指针(或引用)，而函数 f()返回的是该类的引用，因此就没有任何问题。

如果去掉基类中 g()前面的 virtual，这个函数就不是虚函数，这个时候就不会产生编译错误，因为 D 类定义的是一个新函数 g()，隐藏了基类的同签名的函数 g()。

同样，假如 D 里的 g()和 B 里的 g()的函数签名不同，它们就不是同一个虚函数。例如：

```
class B {
public:
    virtual B& f() { return *this; }
```

```
    virtual void  g(int) { std::cout << "g in B\n"; }
};
class D :public B {
public:
    D& f() { return * this; }
    void g() { std::cout << "g in D\n"; }
};
```

D里定义的函数g()是一个新的函数，而不是从基类继承的虚函数g()。也就是说，D里有2个函数g()。

对于下面的代码：

```
int main() {
    D d;
    B *p = &d;
    p->f();
    d.g();
    p->g();
}
```

通过基类指针p调用派生类的g()是错误的，因为该函数不是虚函数，只能通过p调用带一个int类型参数的虚函数g(int)：

```
p->g(0);
```

3. override

在定义派生类时，可能会由于疏忽而写错虚函数名。如：

```
class X {
public:
    virtual void  print() { }
};
class Y :public X {
public:
    virtual void  Print() { }
};
```

将Y类从X类继承的虚函数print()的首字母写成了大写，导致这是2个不同的虚函数。为了避免这种错误，可以通过在派生类的虚函数签名后添加**override**关键字，说明这是一个从基类继承下来的虚函数，编译器会检查基类是否有这个虚函数，如果没有就会报告错误。例如：

```
class Y :public X {
public:
    virtual void  Print() override{ }
};
```

因为基类没有Print()的虚函数，编译器将产生编译错误：

```
error C3668: "Y::Print": 包含重写说明符"override"的方法没有重写任何基类方法
```

override 有助于提早预防这种打字错误。

4. final

虚函数可以一直被派生类继承下去，但有时希望在派生类的某个层次上终止虚函数的向下传递，即一个派生类不允许它的直接或间接派生类继承或定义这个虚函数。此时可以在当前类的虚函数签名后添加 final 关键字，表示虚函数的继承到此为止，其派生类不能再定义或继承该虚函数了。例如：

```
class Y :public X {
public:
    virtual void print() override final{ }
};
```

表示 Y 类的派生类不能再有 print()虚函数。

定义一个类时，可以在类名后用关键字 final 将一个类定义为 final 类(最终类)，即不能再从这个类定义任何派生类。例如：

```
class Y final :public X {
public:
    virtual void  print() override { }
};
```

表示不能从 Y 类定义派生类，或者说任何类不能将 Y 作为基类。

9.4.5 基类指针数组

可以用一个基类指针数组来存储不同派生类对象的指针。例如：

```
int main() {
    Person * arr[5];
    int n = 0;
    arr[0] = new Teacher("Li Ping","讲师");
    arr[1] = new Teacher("张伟", "教授");
    arr[2] = new Student("王浩", 70.5);
    n = 3;
    for (auto i = 0; i != n; i++)
        arr[i]->print();
}
```

该程序用基类指针 Person * 类型的数组 arr 存储不同派生类对象的地址，在循环中通过基类指针调用虚函数 print()输出该指针指向的实际对象的信息。输出结果：

```
Li Ping 讲师
张伟    教授
王浩    70.5
```

9.4.6 虚析构函数

假设给前面的 Person、Teacher、Student 3 个类分别添加了输出如下信息的析构函数：

```
~Person() { cout <<"Person 的析构函数";  }
~Student() { cout <<"Student 的析构函数"; }
~Teacher() { cout <<"Teacher 的析构函数"; }
```

然后在上述 main()函数的最后添加如下语句释放这些动态分配的派生类对象：

```
for (auto i = 0; i != n; i++) delete arr[i];
```

程序输出是：

```
Person 的析构函数
Person 的析构函数
Person 的析构函数
```

这说明调用的都是基类的析构函数，显然是不对的。这是因为析构函数没有定义为虚函数。为了保证正确的释放基类指针指向的派生类对象，应该将基类的析构函数定义为虚函数，自然派生类的析构函数就是虚函数了。例如：

```
virtual ~Person() { cout <<"Person 的析构函数\n";}
```

重新执行程序，可以看到循环语句中的 delete 调用的是派生类的析构函数。输出结果：

```
Teacher 的析构函数
Person 的析构函数
Teacher 的析构函数
Person 的析构函数
Student 的析构函数
Person 的析构函数
```

当然每个派生类的析构函数也隐式调用了其基类 Person 的析构函数。并且先调用派生类的析构函数再调用基类的析构函数，这个次序正好和构造函数调用的次序相反。

9.4.7 纯虚函数和抽象类

函数体＝0 的虚函数称为纯虚函数(pure virtual function)。如表示一个几何形状的 Shape 类的 draw()和 area()都是纯虚函数。

```
#include <iostream>
struct Vector2 {
    double x, y;
};
class Shape {
```

```
    Vector2 pos;
protected:
    Shape(const Vector2& pos) : pos{ pos } {}
public:
    Vector2 position() { return pos; }
    virtual void draw() = 0;
    virtual double area() = 0;
};
```

纯虚函数主要是指那些类不知道怎么实现而需要其派生类实现的函数，这里类 Shape 表示的是一个抽象的形状，抽象形状当然无法确定面积，也不可能绘制它们，将 draw()、area()定义成纯虚函数就表示它们是抽象的方法。

包含纯虚函数的类称为抽象类(abstract class)。抽象类是不能实例化的，即不能定义抽象类的对象。派生类只有实现了基类的所有纯虚函数，才不是一个抽象类，否则，仍然是一个抽象类。如：

```
class Circle:public Shape {
    const inline static double PI{ 3.1415926 };
public:
    Circle(const Vector2& pos, double r) :Shape{pos},radius { r } {}
    double radius{ 0 };
    double area() { return PI * radius* radius; }
};

class SolidCircle :public Circle {
    int color;
public:
    SolidCircle(const Vector2& pos, double r, int cor)
        :Circle{ pos,r }, color{cor}{}
    void draw() {
        std::cout << "draw a sold_circle with radius "
            << radius << std::endl;
    };
};

int main() {
    Vector2 v{ 0,0 };
    Shape s(v);               //错:不能实例化一个抽象类(即不能定义抽象类的对象)
    Circle c(v,1.);           //错:不能实例化一个抽象类(即不能定义抽象类的对象)
    SolidCircle sc(v, 1.,3);
    sc.draw();
}
```

上述代码中的 Shape、Circle 都是抽象类，Ciecle 尽管实现了纯虚函数 area()，但仍然有一个继承自 Shape 的纯虚函数 draw()。因此，不能定义这 2 个类的对象，而 SolidCircle 是一个非抽象类，可以定义 SolidCircle 类的对象。要使 Circle 成为非抽象类，就必须实现其基类 Shape 中的所有纯虚函数。

抽象类的纯虚函数描述了从它派生的派生类都具有的统一的接口。定义抽象类是为了从它定义可实例化的派生类，可以用这个抽象类的指针或引用去指向或引用一个派生类对

象,方便对各种派生类对象进行统一管理。例如可以用一个 shape * 类型的数组统一管理所有不同种类的对象。

尽管不能定义抽象类的对象,但派生类对象构造时会调用基类的构造函数,从而对其基类的抽象类私有成员变量(如 Shape::pos)初始化。因此,对于这个(Shape)抽象类也应该定义构造函数,将其构造函数声明为 protected,进一步明确表示外界无法使用其构造函数去实例化对象。另外,抽象类的构造函数中也不能调用纯虚函数。

9.5 实战:仿"雷电战机"游戏

第 7 章中编写了基于链表的贪吃蛇游戏,本节将开发一个模仿著名的"雷电战机"的射击类游戏。

9.5.1 精灵

贪吃蛇游戏中的物体是可以运动、吃鸡蛋的蛇和随机出现的鸡蛋,乒乓游戏里是可以运动的球、挡板,而雷电战机游戏中将包含敌我双方的战机、子弹等。游戏中这些可以运动或随机出现的物体称为精灵。所有种类的精灵具有一些共同的属性,如位置、速度、颜色等,每种精灵还具有自身的特性,如蛇会移动、吃鸡蛋,而鸡蛋不会运动。

采用面向对象编程的继承思想,可以定义一个精灵类 Sprite 表示所有精灵具有的共同属性,然后从这个一般性的精灵类再派生出各种具体的精灵,如表示战机的 Fighter、表示子弹的 Bullet 等。

这些精灵都是一些具有自主行为能力的对象,它们相互之间可以发送、接收消息(射击、碰撞),也可以接收外界消息(如玩家的键盘输入)。

游戏中经常需要检测精灵之间是否发生了碰撞(collision),一种简单快速的碰撞检测是为每个精灵计算一个包围的矩形(rectangle),然后通过检查矩形之间是否发生相交来检测精灵是否发生了碰撞。为此,在定义精灵类之前,先定义一个表示矩形包围盒的 Rect 类:

```
//包围矩形
using Vector2 = Position;          //Vector2 就是贪吃蛇的 Position 类
class Rect{
public:
      Vector2 pos, size;           //左上角位置和长宽
      Rect(Vector2 p, Vector2 s) :pos{ p }, size{ s }{}
      Rect(const T x,const T y, const T w, const T h)
            :pos{x,y}, size{w,h}{}

      bool collide(const Rect &other) {
            return (
                  (pos[0] < other.pos[0] + other.size[0]) &&
                  (pos[0] + size[0] > other.size[0]) &&
                  (pos[1] < other.pos[1] + other.size[1]) &&
                  (pos[1] + size[1] > other.size[1])
                  );
```

```
        }
};
```

其中,collide()函数可以检查2个Rect对象表示的矩形是否碰撞(相交)。

现在,可以定义一个如下的精灵类:

```
//精灵
class Sprite {
protected:
        Window * window{ nullptr };
        Vector2 pos, vel,size_;      //位置、速度、大小。假设 pos 是精灵的中心点位置
        Color color;                 //颜色
        int lives{1};                //生命值
        Rect rect;
public:
        Sprite(Window * window,const Color c, const Vector2 p,
                const Vector2 v = Vector2{0,0},const  Vector2 s = Vector2{ 1,1 },
                const int lives = 1)
                :window{ window },pos {      p}, vel{ v }, size_{ s },
                color{ c }, lives(lives), rect{ Vector2(pos[0] - s[0]/2,pos[1] - s[1]/2),s }
        {
        }
        virtual void update() {      //根据速度更新位置
                pos[0] += vel[0]; pos[1] += vel[1];
                rect.pos[0] = pos[0] - rect.size[0] / 2;     //计算包围矩形的左上角位置
                rect.pos[1] = pos[1] - rect.size[1] / 2;
        }
        virtual void draw() {}
        virtual bool is_dead() { return lives <= 0; }
        virtual Rect get_rect() { return rect; }        //返回包围矩形
        virtual bool collide(const Rect &other) { return rect.collide(other); }
};
```

每个精灵有一个update()函数用于更新其状态,还有一个draw()函数用于在Window画布上绘制自己的图像。注意,这些函数都是虚函数。

从这个一般的精灵类可以派生出各种具体的精灵,如敌方战机、我方战机、子弹等。

首先定义一个简单的子弹类:

```
class Bullet:public Sprite {
public:
        Bullet(Window * window, const Color c, const Vector2 p,
                const Vector2 v = Vector2{ 0,0 }) :Sprite(window,c,p,v) {
        }
};
```

子弹类Bullet和一般的精灵类目前没有区别。

我方战机、敌方战机都具有所有战机的共同属性,如射击shot()。因此,可先定义一个战机类**Flighter**:

```
class Flighter:public Sprite {
    protected:
        Bullet * bullet{nullptr};
    public:
        Flighter(Window * window, const Color c, const Vector2 p,
            const Vector2 v = Vector2{ 0,0 }, const  Vector2 s = Vector2{ 1,1 },
            const int lives = 1) :Sprite(window, c, p, v, s, lives) {
        }
        //根据速度 vel 发射一颗子弹
        virtual Bullet * shot(const Vector2 &vel) {
            auto x = pos[0];
            auto y = pos[1] - rect.size[1] / 2;
            bullet = new Bullet(window, bullet_color, Vector2{ x,y }, vel);
            return bullet;
        }
};
```

战机类 Flighter 有一个私有变量 bullet 保存射出的子弹，还有一个成员函数 shot()用于射击(即发出一颗子弹)。

然后从 Flighter 类派生出我方战机类 Player：

```
class Player :public Flighter{
public:
    Player(Window * window, const Color c, const Vector2 p,
        const Vector2 v = Vector2{ 0,0 }, const  Vector2 s = Vector2{ 1,1 },
        const int lives = 1) :Flighter(window, c, p, v, s, lives) {
    }
    void move(Vector2 vel) {
        auto x = pos[0] + vel[0];
        auto y = pos[1] + vel[1];
        if (x >= rect.size[0]/2&&x < window->get_width() - rect.size[0]/ 2 - 1)
            pos[0] = x;
        if (y >= rect.size[1]/2 && y < window->get_height() - rect.size[1]/2 - 1)
            pos[1] = y;
    }

    virtual void draw() {
        auto x{pos[0]}, y {pos[1]};
        window->set_pixel(x, y - 1, player_color);
        window->set_pixel(x - 1, y, player_color);
        window->set_pixel(x, y, player_color);
        window->set_pixel(x + 1, y, player_color);
        window->set_pixel(x - 1, y + 1, player_color);
        window->set_pixel(x + 1, y + 1, player_color);
    }
};
```

类 Player 具有一个 move()函数表示运动，并重新定义了 draw()函数用于绘制我方战

机的图像。

同样地，可以定义表示敌方战机的类 Enemy 及其 draw()虚函数。

```
class Enemy :public Flighter {
public:
    Enemy(Window * window, const Color c, const Vector2 p,
        const Vector2 v = Vector2{ 0,0 }, const  Vector2 s = Vector2{ 1,1 },
        const int lives = 1) :Flighter(window, c, p, v, s, lives) {
    }
    virtual void draw() {
        auto x{ pos[0] }, y{ pos[1] };
        window->set_pixel(x-2, y - 1, enemy_color);
        window->set_pixel(x , y - 1, enemy_color);
        window->set_pixel(x + 2, y - 1, enemy_color);
        window->set_pixel(x - 1, y , enemy_color);
        window->set_pixel(x+1, y , enemy_color);
        window->set_pixel(x , y+1, enemy_color);
    }
};
```

9.5.2 游戏引擎 GameEngine

首先改写一下第 7 章的针对任何游戏的游戏引擎类 GameEngine，以便将一系列精灵保存在一个线性表中。

```
//首先将表示线性表的 Vector 的数据元素类型 ElemType 定义为 Sprite*
using ElemType = Sprite*;
//using ElemType = int;
class Vector{
    ElemType *data{ nullptr };
    int  capacity{ 0 }, n{ 0 };
public:
    Vector(const int cap=5) :capacity{ cap }, data{ new ElemType[cap] }
    {}
    bool insert(const int i, const ElemType &e) {
        if (i<0 || i>= n) return false;
        if (n == capacity&&!add_capacity())
                return false;
        for (auto j = n; j>i; j--)
            data[j] = data[j - 1];
        data[i] = e;
        return true;
    }
    bool erase(const int i) {
        if (i<0 || i>= n) return false;
        for (auto j = i; j<n-1 ; j++)
            data[j] = data[j+1];
        n--;
        return false;
```

```
    }
    bool push_back(const ElemType &e) {
        if (n == capacity && !add_capacity())
            return false;
        data[n++] = e;
        return true;
    }
    bool pop_back(const ElemType e) {
        if (n == 0) return false;
        n--; return true;
    }

    ElemType get(const int i)const {
        if(i>=0&&i<n)
            return data[i];
        throw "下标非法!\n";
    }
    ElemType get(const int i,const ElemType &e)const {
        if (i>= 0 && i<n)
            return data[i] = e;
    }

    ElemType& operator[](int i) {
        if (i>= 0 && i<n) return data[i];
        else throw "下标非法";
    }
    ElemType operator[](const int i) const {
        if (i>= 0 && i<n) return data[i];
        else throw "下标非法";
    }

    bool add_capacity(){
        ElemType *temp = new ElemType[2 * capacity];
        if (!temp) return false;
        for (auto i = 0; i<n; i++) {
            temp[i] = data[i];
        }
        delete[] data;
        data = temp; capacity *= 2;
        return true;
    }

    int size() const{ return n; }
};
//----------游戏引擎类------------------
class GameEngine {
protected:
    Window *window{ nullptr };
    Vector sprites;              //所有精灵对象指针的线性表
    bool running{ true };
    BackGround *bg{ new BackGround() };
```

```
    public:
        GameEngine(const int w = 80, const int h = 30) {
            window = new Window(w, h, ' ');
            window->clear();
        }

        virtual void run() {
            while (running) {
                processEvent();
                update();
                collision();
                render();
            }
            quit();
        }

        virtual void processEvent() {
            //处理事件
            char key;
            if (_kbhit()) {
                key = _getch();
                if (key == 27) running = false;
            }
        }
        virtual void update() {
            for (auto j = 0; j < sprites.size(); j++)
                sprites[i]->update();
        }
        virtual void collision() {

        }
        virtual void render() {
            if (!running) return;
            gotoxy(0, 0);
            hideCursor();
            window->clear();
            draw_scene();
            window->show();
        }
        virtual void draw_scene() {
            bg->draw(*window);
            for (auto j = 0; j < sprites.size(); j++)
                sprites[i]->draw();
        }
        virtual void quit() {}
    };
```

即用一个 Sprite * 指针的 std::vector<>管理游戏中的所有精灵。注意,GameEngine 类的函数都是虚函数。其中的绘制场景函数 draw_scene()调用每个精灵的自身的绘制函数 draw()。而 update()也调用每个精灵自身的 update()更新自己的状态(数据)。

然后可以从这个一般性的游戏引擎类派生出定义针对雷电战机的游戏引擎类SpaceInvander：

```
#define KEY_UP 72
#define KEY_DOWN 80
#define KEY_LEFT 75
#define KEY_RIGHT 77

class SpaceInvander :public GameEngine {
protected:
    Player *player{nullptr};
    Vector enemies;              //敌机的线性表
    Vector bullets;              //子弹的线性表
public:
    SpaceInvander(const int w = 80, const int h = 30) : GameEngine(w,h){
        auto player_w{3 }, player_h{ 3 };
        player = new Player(window, player_color,
            Vector2{ window->get_width()/2,window->get_height() - player_h-1 },
            Vector2{ 0,0 }, Vector2{ player_w,player_h });
        sprites.push_back(player);

        //生成位置随机的敌机
        int x_off = 10,y_off = 5;
        auto x_min{ x_off }, x_max{ window->get_width() - x_off },
            y_min{ 1 }, y_max{ y_off };
        auto x = random_int(x_min, x_max);
        auto y = random_int(y_min, y_max);
        Enemy *enemy = new Enemy(window, enemy_color, Vector2{ x,y });
        sprites.push_back(enemy);
        enemies.push_back(enemy);
    }

    void processEvent() {
        char key;
        if (_kbhit()) {
            key = _getch();
            if (key == 27) running = false;
            if (key == ' ') {     //空格键表示发射子弹
                //生成子弹的位置正好在战机的上方
                Bullet *bullet = player->shot(Vector2(0,-1));
                bullets.push_back(bullet);
                sprites.push_back(bullet);
            }
            else if (key == 'a'||key == 'A'||key== KEY_LEFT) {
                player->move(Vector2(-1, 0));       //战机左移
            }
            else if (key == 'd'||key == 'D'||key == KEY_RIGHT) {
                player->move(Vector2(1, 0));        //战机右移
            }
            else if (key == 'w'||key == 'W'||key == KEY_UP) {
```

```
                player->move(Vector2(0,-1));
            }
            else if (key == 's' || key == 'S' || key == KEY_DOWN) {
                player->move(Vector2(0,1));
            }
        }
    }
};
```

最后在 main()函数中初始化一个 SpaceInvander 对象，并调用 run()运行该游戏：

```
int main() {
    SpaceInvander game;
    game.run();
}
```

执行该程序，在画面上出现我方战机和敌方敌机，如图 9-3 所示。用键盘的'w'、's'、'a'、'd'和方向箭头键可以控制其上、下、左、右移动，按空格字符，就可以射出一系列的子弹。

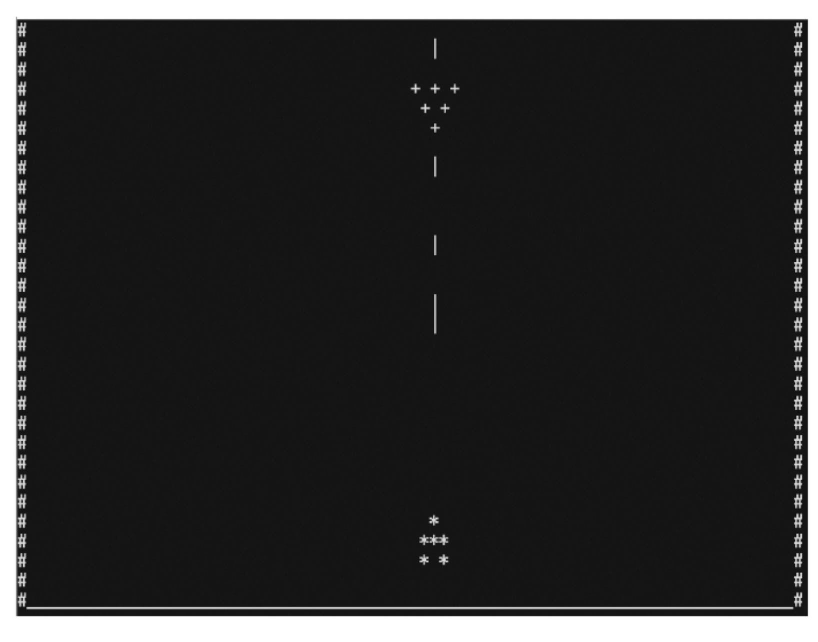


图 9-3　我方战机和敌方战机

目前程序有一些问题：首先缺少碰撞检测，当子弹击中对方或敌我双方战机碰撞时，怎么办？另外，当子弹或敌方战机超出窗口画面时，这些子弹和敌方战机不会再出现了，应该销毁这些对象。

9.5.3　碰撞检测和精灵的销毁

可以在游戏引擎类的 collision()函数中处理精灵的碰撞检测和销毁。

（1）碰撞检测：对每个敌方战机，检测它是否和我方战机碰撞，检测是否被某个子弹击中；对于我方战机，检测其是否被子弹射中。

(2) 精灵的销毁：对于每个死亡的精灵，除了将它的指针在 sprites 中和特定类型的数组如 enemies 或 bullets 中删除，还要在内存中销毁该对象以释放它们占用的内存等资源。

```
void collision() {
    //检测是否碰撞
    //检测敌方战机是否被子弹击中或和我方战机碰撞
    for (auto i = 0; i < enemies.size(); i++) {        //对每个敌方战机
        auto enemy = enemies[i];
        if (player&&player->collide(enemy->get_rect())) {
            player->hitted();
            static_cast<Enemy*>(enemy)->hitted();
        }
        for (auto j = 0; j < bullets.size(); j++) {    //对每颗子弹
            auto bullet = bullets[j];
            if (enemy->collide(bullet->get_rect())) //敌方战机和子弹是否碰撞
                static_cast<Enemy*>(enemy)->hitted();
                static_cast<Bullet*>(bullet)->hitted();
            }
        }
    }
    //检测我方战机是否被子弹击中
    if (player) {
        for (auto j = 0; j < bullets.size(); j++) {
            auto bullet = bullets[j];
            if (player->collide(bullet->get_rect()))//被子弹 bullet 击中
                player->hitted();
                static_cast<Bullet*>(bullet)->hitted();
            }
        }
    }
    Vector deads;                                      //记录死亡精灵的 Vector
    for (auto i = 0; i < sprites.size();) {
        if (sprites[i]->is_dead()) {
            delete sprites[i];               //删除该精灵
            deads.push_back(sprites[i]);     //精灵指针放入死亡的 Vector 中
            sprites.erase(i);                //在 sprites 中删除该精灵的指针
        }
        else i++;
    }
    //在 enemies 和 bullets 中寻找已经死亡的 Sprite 指针
    for (auto i = 0; i < deads.size();i++) {
        Sprite* p = deads[i];                //死亡精灵的指针
        auto deleted{ false };
        for (auto i = 0; i < enemies.size(); i++)  //死亡精灵是否是一个敌方战机
            if (enemies[i] == p) {           //如果在 enemies 中，则从中删除
                enemies.erase(i);
                deleted = true;break;        //跳出 for 循环
            }
        if (deleted) continue;               //已经删除了这个死亡精灵的指针
        for (auto i = 0; i < bullets.size(); i++)    //死亡精灵是否是一颗子弹
```

```
            if (bullets[i] == p) {
                bullets.erase(i);
                break;
            }
        }
}
```

9.5.4　让敌方战机运动和发射子弹

目前，敌方战机既不能移动，也不能发射子弹。

可以让敌方战机周期性地发射子弹，为此需要能得到当前的时间，可以用头文件 chrono 的时间函数，即包含该头文件：

```
#include <chrono>
```

修改 Enemy 的 update()函数使得定时开始发射子弹，如上次开始发射子弹后经过 2000ms 就开始新一轮的发射子弹，每次发射 5 颗子弹，但每颗子弹之间也需要有一定的时间延迟，不然子弹就连续成一条线了，即当最后一次发射子弹并经过了一段时间，如 100ms，就开始发射第二颗子弹：

```
virtual void update() {
    //随机发射子弹
    static int bullet_num{ 5 };                  //假如每次连续发射 5 发子弹
    static auto shot_start = std::chrono::high_resolution_clock::now();
                                                 //上次开始发射子弹的时间
    static auto shot_last = shot_start;        //最后一颗子弹的时间
    auto now = std::chrono::high_resolution_clock::now();
                                                 //当前时间
    auto dur = now - shot_start;                 //距离上次发射子弹的时间间隔
    auto ms = std::chrono::duration_cast<std::chrono::milliseconds>(dur).count();
    if (ms > 2000) {                             //时间间隔超过 2000ms
        bullet_num = 3;
        shot_start = now;
    }
    if (bullet_num > 0) {
        auto dur2 = now - shot_last;             //距离最后一个子弹的时间间隔
        auto ms2 = std::chrono::duration_cast<std::chrono::milliseconds>(dur2).count();
        if (ms2 > 100) {                         //超过 100ms，发射新的子弹
            shot_last = now;
            shot(Vector2{ 0,1 });                //必须将这个子弹在引擎类的 update()中加入
            bullet_num--;                        //剩余子弹数目
        }
    }
    Flighter::update();
}
```

为了将发射的子弹加到引擎类中，提供一个辅助函数：

```
Bullet * get_bullet(){                    //返回 bullet 指针，并将 bullet 设置为空指针
    Bullet *p = bullet;
    bullet = 0;
    return p;
}
```

相应地，SpaceInvander 的 update()函数要做修改，以便将这个周期性发射的子弹加到精灵列表和子弹列表中。

```
class SpaceInvander :public GameEngine {
protected:
    //…
    Vector bullets, enemy_bullets;
public:
  //…
      void update() {
          for (auto i = 0; i < enemies.size(); i++) {
              auto enemy = enemies[i];
              Bullet * bullet = static_cast<Enemy *>(enemy) -> get_bullet();
              if (bullet) {  //如果发射了子弹，这个子弹的指针加入 enemy_bullets 和
                             //sprites 中
                  enemy_bullets.push_back(bullet);
                  sprites.push_back(bullet);
              }
          }
          GameEngine::update();
      }

      void collision() {
          //…
          //检测我方战机是否被子弹击中
          if (player) {
              for (auto j = 0; j < enemy_bullets.size(); j++) {
                  auto bullet = enemy_bullets[j];
                  if (player -> collide(bullet -> get_rect())) {
                      player -> hitted();
                      static_cast<Bullet *>(bullet) -> hitted();
                  }
              }
          }

          //…
          //在 enemies 和 bullets 中寻找已经删除的 Sprite 指针
          for (auto i = 0; i < deads.size();i++) {
              //…
              if (deleted) continue;
              for (auto i = 0; i < enemy_bullets.size(); i++)
                  if (enemy_bullets[i] == p) {
```

```
                        enemy_bullets.erase(i);
                        break;
                    }
                }
            }
        };
```

再次运行 main()函数，此时可以看到敌方战机会定时发射一串子弹。如果我方战机被击中就被销毁了。

和随机发射子弹一样，可以让敌方战机随机向前或左右运动，在之前的 Enemy 类的 update()的发射子弹代码后面添加如下代码：

```
static auto move_start = std::chrono::high_resolution_clock::now();
auto dur_move = now - move_start;             //持续时间
auto ms_move = std::chrono::duration_cast<std::chrono::milliseconds>(dur_move).count();
if (ms_move > 300) {
    auto a = ms_move % 6;
    if (a > 0) {
        if (a == 1) move(Vector2(-1, 0));
        else if (a == 2) move(Vector2(1, 0));
        else if (a > 2)move(Vector2(0, 1));
        move_start = now;
    }
}
```

再次运行 main()函数，可以看到敌方战机会随机地前进或左右运动，如图 9-4 所示。



图 9-4　敌方战机会随机地前进或左右运动

上述程序还有许多问题和需要改进的地方：

- 当发射碰撞或战机被击中时，缺少爆炸效果。
- 敌方战机和子弹的种类单一，武器是否可以升级？
- 没有显示得分或损失情况。
- 敌方战机是否可以追击撞击我方战机？

作为练习，希望读者能进一步完善这个雷电战机游戏。

9.6 习题

1. 解释 override 和 final 关键字的作用。何时应该使用它们？定义类成员函数时是否可以同时使用它们？

2. 虚函数和纯虚函数有什么区别？何时应该使用虚函数或纯虚函数？

3. 下面程序的输出是什么？

```
#include <iostream>
class B {
public:
    void print() { std::cout << "print in B\n"; }
    virtual void hello() { std::cout << "hello in B\n"; }
};

class D:public B {
public:
    void print() { std::cout << "print in D\n"; }
    void hello() { std::cout << "hello in D\n"; }
};
int main() {
    B b; D d;
    b.print();d.print();
    b.hello();d.hello();
    B * bp = &b;
    bp->print(); bp->hello();
    bp = &d;
    bp->print(); bp->hello();
    D * dp = &d;
    dp->print(); dp->hello();
    dp = static_cast<D*>(&b);
    dp->print(); dp->hello();
}
```

4. 下面程序的输出是什么？

```
#include <iostream>
using namespace std;

class A{
public:
```

```
    A() { cout << "A"; }
    A(const A &) { cout << "a"; }
};
class B : public virtual A{
public:
    B() { cout << "B"; }
    B(const B &) { cout << "b"; }
};
class C : public virtual A{
public:
    C() { cout << "C"; }
    C(const C &) { cout << "c"; }
};
class D :B, C{
public:
    D() { cout << "D"; }
    D(const D &) { cout << "d"; }
};
```

5. 解释下面程序的输出结果。

```
#include <iostream>
struct A {        int a;};
struct B : virtual A{        int b;};
struct C : virtual A {        int c;};
struct D : B,C {        int d;};
int main() {
    std::cout << sizeof(A) << '\t';
    std::cout << sizeof(B) << '\t';
    std::cout << sizeof(C) << '\t';
    std::cout << sizeof(D) << '\n';
}
```

6. 下面程序的输出是什么?

```
int main(){
    D d1;
    D d2(d1);
}
#include <iostream>
struct A {
    virtual std::ostream &put(std::ostream &o) const {
        return o << 'A';
    }'
};
struct B : A {
    virtual std::ostream &put(std::ostream &o) const {
        return o << 'B';
    }
```

```
};
std::ostream &operator <<(std::ostream &o, const A &a) {
    return a.put(o);
}
int main() {
    B b;
    std::cout << b;
}
```

7. 下面程序的输出是什么?

```
#include <iostream>
struct Base {    virtual int f() = 0;};
int Base::f() { return 1; }

struct Derived : Base {    int f() override;};
int Derived::f() { return 2; }
int main() {
    Derived object;
    std::cout << object.f();
    std::cout << ((Base&)object).f();
}
```

8. 下面程序的输出是什么?

```
#include <iostream>
struct X{
    X() { std::cout << "1"; }
    X(const X&) { std::cout << "2"; }
    ~X() { std::cout << "3"; }
};
X f(){
    X x;   return x;
}
int main(){  f();}
```

9. 下面程序的输出是什么?

```
#include <iostream>
class A {
public:
    void f() { std::cout << "A"; }
};
class B : public A {
public:
    void f() { std::cout << "B"; }
};
void g(A &a) { a.f(); }
int main() {B b;g(b);}
```

10. 下面程序的输出是什么？

```
#include <iostream>
struct A{
    virtual int foo(int x = 5){
    return x * 2;       }
};

struct B : public A{
    int foo(int x = 10){
        return x * 3;       }
};

int main(){
    A* a = new B;
    std::cout << a->foo() << std::endl;
    return 0;
}
```

提示：通过指针指向的或引用调用的虚函数的默认参数的值是由指针或引用的类型的虚函数决定的，而不是由其实际对象的类型的虚函数决定的。

11. 下面代码的错误是什么？请改正之。

```
#include <iostream>
using std::cout;
struct Base {
    void print(){   cout <<"B\n";    }
};
struct Derived: Base {
    void print(){   cout <<"D\n";    }
};
int main(){
    Base* b1 = new Base;
    if(Derived* d = dynamic_cast<Derived*>(b1)) {
        std::cout << "downcast from b1 to d successful\n";
        d->print();
    }
     Base* b2 = new Derived;
    if(Derived* d = dynamic_cast<Derived*>(b2)) {
        std::cout << "downcast from b2 to d successful\n";
        d->print();
    }
    delete b1;
    delete b2;
}
```

12. 定义一个表示二维几何形状的抽象类 Shape，该类只有 2 个纯虚函数 draw()和area()，然后从该类派生出表示圆的 Circle 类、表示矩形的 Rectangle 类和表示三角形的 Triangle 类，每个具体的形状类具有不同的属性，如圆有圆心和半径、矩形有表示左下角位

置的坐标和长宽、三角形有 3 个顶点坐标。假设 draw()成员函数用输出信息来模仿绘制图形行为而不是真正在图形环境下绘制。如：

```
void Circle::draw() {
    std::cout << "绘制坐标在(" << x << "," << y << ")的圆\n";
}
```

编写一个程序，可以从键盘输入不同的形状参数，用动态内存分配方法创建这个形状对象，并将其指针保存在一个 Shape * 数组或线性表中。程序能通过遍历这个数组或线性表的每个指针，调用绘制函数 draw()绘制。

13. 在表示日期的 Date 类和表示顺序表的 Vector 类基础上，定义一个表示雇员的类 Employee，包含姓名、雇用日期和部门编号等信息，然后从 Employee 类派生出一个表示经理的类 Manager。经理除继承雇员的属性外，还包含经理的级别(level)和管理的一组雇员，并在此基础上写一个简单的公司员工管理程序。要求该程序具有录入新员工，显示所有员工，查询、修改和删除员工等功能。

14. C 风格强制类型转换、static_cast 和 dynamic_cast 的应用场景是什么？在 9.5.3 节的代码里为什么用 static_cast 而不用 dynamic_cast？



模　　板

C++的强大的模板是**通用算法（泛型编程）**的基础。

10.1 函数模板

下面是一个针对 int 类型的数组排序的函数：

```
void sort(int arr[], int n){
    //...
}
```

这个函数只能对 int 类型的数组进行排序，如果需要对其他类型如 double 或用户定义类型如 Student 的数组排序，怎么办？通常会采用复制、粘贴、修改的方法，将针对 int 类型数组的函数修改成针对其他不同数据元素类型的数组的函数，如：

```
void sort(double arr[], int n){
    //...
}
void sort(Student arr[], int n){
    //...
}
```

可以分别用来对 double 或 Student 类型的数组进行排序。

但实际问题中的数据元素类型是各种各样的，如果针对每种类型都要重新编写一份"除数据类型不同外，代码完全一样"的函数，不仅单调乏味，也容易因为某处忘记修改而带来隐藏的错误，另外，将来如果发现代码需要修改，则所有这些函数都要做相应的修改。

再看一个更简单的例子，假如要求 2 个对象的最大值，如对 2 个 int 类型对象，可写出如下函数：

```
int Max(int a,int b) {
```

```
    return a > b?a:b ;
}
```

对两个 double 类型对象,可写出如下代码:

```
double Max(double a,double b) {
    return a > b?a:b ;
}
```

除了数据元素类型不同外,这 2 个 Max()函数的代码是完全一样的。但这 2 个 Max()函数只能分别用于对 int 和 double 类型的对象求最大值,而不能用于其他数据类型的对象。

针对这个问题,C++提供了**函数模板(function templates)**。通过将一个函数写成一个函数模板,可以自动从这个函数模板生成针对具体数据类型的具体函数。C++标准库中大量使用了函数模板以保证算法可以适用于任何数据类型,包括那些将来程序员可能定义的各种未知数据类型。

10.1.1 函数模板的定义与实例化

为了解决这种需要重复编写"数据元素的类型不同而代码完全相同"的不同重载函数,C++通过**模板(template)**提供了编写通用代码的手段。对上述例子,可以编写如下 Max 模板:

```
template < typename T >
T Max(T a,T b) {
    return a > b?a:b ;
}
```

上面的代码定义了一个叫作 Max 的**函数模板**,其中 **template < typename T >**称为**模板头**,以关键字 template 开头,三角箭头<>之间的部分称为**模板参数**,其中以 typename 声明了一个**模板类型参数 T**。函数模板签名中,在圆括号()中的是类似普通函数的函数形参列表,其中的形参 a、b 的数据类型就是模板类型参数 T。Max()函数模板名前的 T 表示**返回类型**,也是模板类型参数 T。

函数模板本身并不是一个具体的函数,而是用于产生具体函数的蓝图。如同生产零件的模具或设计图本身不是一个具体产品、类本身不是一个具体的对象,而所有对象的蓝图都一样。有了上面的函数模板,可以通过给函数模板指定实际的模板参数来**实例化**一个函数模板,即产生一个具体的函数:

```
int main(){
    cout << Max < int >(3, 5) << endl;          //模板类型参数 T 为 int
    cout << Max < double >(3.5, 4.5) << endl;   //模板类型参数 T 为 double
    cout << Max < int >(6, 4) << endl;
}
```

其中,Max < int >通过给函数模板 Max 的模板类型参数 T 传递实际的模板参数 int,即指定

三角箭头中的模板类型参数 T 为 int,产生了一个具体的数据类型为 int 的一个 Max()函数,编译器针对 Max<int>会自动生成如下函数。

```
int Max(int a,int b) {
    return a > b?a:b ;
}
```

而不需要程序员自己编写这个参数为 int 类型的 Max()函数。即 **Max<int>(3, 5)**先生成一个 int 类型的 Max()函数,然后将实际参数 3、5 传给生成的这个 Max()函数 int Max(int a,int b)的 2 个形参 a、b。

从函数模板生成的具体函数称为这个模板的一个**实例**,这个过程称为**模板的实例化**。

同样,Max<double>会自动生成模板类型参数是 double 的一个模板实例,即如下的函数:

```
double Max(double a,double b) {
    return a > b?a:b ;
}
```

编译 **Max<double>(3.5, 4.5)**时就会生成类似上面的函数,再将实际参数 3.5 和 4.5 传递给函数 double Max(double a,double b)的 2 个函数形参 a、b。即调用了函数 double Max(double a,double b)。

那么,Max<int>(6,4)会不会再产生一个 int 版本的函数 int Max(int a,int b)呢?实际上,因为编译器已经产生了 int 版本的函数,就不会再产生这个函数了,否则,函数不是重定义了吗?也就是说,对每种数据类型,编译器只在第一次遇到实例化代码如 Max<int>时产生这种类型对应的函数,后续的 Max<int>(6,4)仅仅调用之前已**实例化**的 int 版本的函数。

10.1.2 模板参数推断

上面的函数模板的不同**实例化** Max<int>或 Max<double>,都显式地指定了模板类型参数 T 为 int 或 double。实际上,编译器可以根据调用函数模板传递的实际函数参数(注:不是模板参数)自动推断出模板参数的类型。即上述 main()函数可以写成:

```
int main(){
    cout << Max(3, 5) << endl;          //根据实际参数 3,5 是 int 类型推断出需要实例
                                        //化 Max<int>函数
    cout << Max(3.5, 4.5) << endl;      //根据实际参数 3.5, 4.5 是 double 类型推断出需
                                        //要实例化 Max<double>函数
    cout << Max(6, 4) << endl;          //调用前面已经实例化的 Max<int>函数
}
```

这种根据实际函数参数推断出模板参数的过程叫作**模板参数推断**(**template argument deduction**)。

这种自动的模板参数推断使得代码更加简洁,但有时如果不能自动推断,仍然必须显式

实例化函数模板。例如：

```
cout << Max(3,4.5);
```

因为 2 个参数类型分别是 int 和 double，而函数模板中的函数形参 a、b 是同一种类型，因此编译器无法推断出模板类型参数 T 的类型到底是 int 还是 double。这个情况下，就必须**显式实例化**：

```
cout << Max<double>(3,4.5);
```

明确告诉编译器，模板类型参数是 double。

注意：函数模板头<>的模板参数和函数签名()里的函数参数是 2 个不同的概念。前者是用来实例化一个函数模板以便生成一个具体的函数，而后者是传递给生成的具体函数的参数。另外函数模板头中的模板参数不一定都是表示数据类型的模板类型参数，还可能是模板非类型参数。

注：C++标准库中实际上已经有了类似的函数模板 std::max 和 std::min，分别用于得到两个量的最大值和最小值。感兴趣的读者可以用不同的数据类型测试一下它们。

10.1.3 模板专门化

对于下面的程序：

```
#include <iostream>
template <typename T>
T Max(T a, T b) {
    return a > b ? a : b;
}

int main() {
    int x = 10, y = 20;
    int *p = &x, *q = &y;
    std::cout << *Max(p, q) << '\n';
}
```

将 int * 类型的指针变量 p 和 q 作为函数参数传递给函数模板 Max，产生了一个数据类型是 int * 的实例化函数：

```
int* Max(int * a, int* b) {
    return a > b ? a : b;
}
```

这个函数实际是对两个指针(地址)进行比较，也就是实际比较的是变量 x 和 y 的内存地址，而不是这 2 个指针指向的对象，返回值也是一个地址，所以需要在前面用 * 运算符(*Max(p, q))得到这个指针指向的 int 类型变量的值，因此，程序运行结果可能是：

```
10
```

如何使传递指针变量作为函数参数，但实际比较的是它们指向的变量的值呢？办法是**模板的专门化**，即针对这个函数模板，再定义一个处理特殊情况的专门的函数模板，这个专门的函数模板中的模板参数（如上面的模板类型参数 T）是一个特殊的值（如 int *）。如：

```
template <>
int * Max<int *>(int* a, int* b) {
      return *a > *b ? a : b;
}
```

是针对 int * 类型的 Max 模板的专门化，它以对应的模板类型参数为空的模板头 template <> 开始，然后在模板名的三角箭头中指定具体的模板类型参数 **Max < int** * >，从而定义了一个专门化的 Max 模板。

再次执行程序，输出结果：

```
20
```

即此时 * Max(p, q)使用的是这个专门化函数模板去实例化一个 int * 类型的函数：

```
int * Max(int* a, int* b) {
      return *a > *b ? a : b;
}
```

模板专门化就是一种“特事特办”。对于通常的情况使用的是通用的函数模板，而对于特殊情况使用的就是专门化的函数模板。

10.1.4　函数模板和重载

可以定义和函数模板名同名的函数或模板，例如：

```
template <typename T>
T Max(T a, T b) {
    return a > b ? a : b;
}

int* Max(int * a, int* b) {
    return *a > *b ? a : b;
}

template <typename T>
T Max(T arr[], int n) {
    T ret{ arr[0] };
    for (int i = 1;i!= n;i++)
        if (arr[i]> ret) ret = arr[i];
    return ret;
```

```
}

template<typename T>
T* Max(T* a, T* b) {
    return *a > *b ? a : b;
}
```

上述代码定义了名字为 Max 的 3 个函数模板和一个普通函数。函数模板 template <typename T> T * Max(T arr[], int n)和 template <typename T> T * Max(T * a, T * b)具有不同的参数列表,因此都是不同于 template <typename T> T Max(T a, T b)的新函数模板。

template <typename T> T * Max(T * a, T * b)不是函数模板 template <typename T> T Max(T a, T b)的专门化,而是具有不同参数列表的新的模板,即只能实例化函数形参是指针类型的函数,即当调用 Max(x,y)时,如果 x、y 都是指针类型,就会用这个模板实例化一个指针类型的函数。

和普通的函数重载一样,编译器会根据模板参数或函数参数确定最匹配的实例化函数。如:

```
int main() {
    int x{ 10 }, y{ 20 };
    cout << Max(x, y) << '\n';                  //template<typename T> T Max(T a, T b)
    cout << *Max(&x, &y) << '\n';               //int* Max(int * a, int* b)
    double ds[]{ 3.1,0.2,4.6,7.8 };
    cout << Max(ds, std::size(ds))<<'\n';       //template<typename T> T Max(T arr[], int n)
}
```

当普通函数和针对指针类型的模板都能精确匹配 * Max(&x, &y)时,普通函数优先。即 int * Max(int * a, int * b)优先于 template <typename T> T * Max(T * a, T * b)。可以通过往普通函数里添加一条输出语句来验证这一点:

```
int* Max(int * a, int* b) {
    cout << "呵呵\n";
    return *a > *b ? a : b;
}
```

执行程序,输出结果:

```
20
呵呵
20
7.8
```

10.1.5 模板的返回类型推断

假如有一个函数模型用于对不同类型的两个量进行运算:

```
template < typename T1,typename T2 >
?  add(T1 a, T2 b) {
    return a + b;
}
```

该函数模板的返回类型应该是什么呢？对于下面的代码：

```
add(2,3.5)
```

到底返回类型是 int 还是 double 呢?

为了解决这个问题，从 C++14 开始，可以用函数的返回类型推断的方法从函数返回表达式的结果类型推断函数的返回类型，即可以用 **decltype** 关键字，从一个表达式 exp 中用 decltype(exp)推断出表达式 exp 的结果的类型。对于上述模板，可以采用**尾返回类型**(**trailing return type**)语法，即在函数签名后用—> decltype(exp)来推断模板函数的返回类型，在函数签名前用 auto 关键字：

```
template < typename T1,typename T2 >
auto add(T1 a, T2 b) -> decltype(a + b) {
     return a + b;
}
```

该函数用 decltype(a + b)来推断模板函数的返回类型。还有一个更加简化的写法，就是直接在函数签名前用 decltype(auto)来推断函数的返回类型：

```
template < typename T1, typename T2 >
decltype(auto) add(T1 a, T2 b) {
     return a + b;
}
```

decltype(auto)和 auto 的区别是：auto 推断的总是一个值，而 decltype(auto)推断的类型可以保留原始的类型(如引用类型或 const)。

10.1.6　非类型模板参数

模板头用 typename 声明的模板参数是**模板类型参数**(也叫**类型模板参数**)，即它们代表的是一种**数据类型**，模板实例化时需要传递某个数据类型给这种类型模板参数。模板参数里还可以有**非类型模板参数**，模式实例化时传递给非类型模板参数的是普通的值而不是数据类型。

例如，下面的函数模板中包含了 2 个非类型模板参数 lower 和 upper，用来表示数组的范围：

```
template < typename T, int lower, int upper >
T sum(const T arr[])
{
```

```
    T ret{ arr[lower] };
    for (auto i = lower + 1; i <= upper; i++)
        ret += arr[i];
    return ret;
}
```

可以使用如下该函数模板：

```
int main() {
    int arr[]{ 1,2,3,4,5 };
    std::cout << sum< int, 0, 4 >(arr)<<'\n';
    std::cout << sum< int, 1, 3 >(arr) << '\n';
    double arr2[]{ 1.2,2.2,3.4,4.7,5.8 };
    std::cout << sum< double, 1, 3 >(arr2) << '\n';
}
```

即在实例化模板时，非类型模板参数传递的是具体的数值，如0、4，而不是数据类型。执行程序运行，输出结果：

```
15      9       10.3
```

可以将函数模板的类型模板参数放在最后，其类型由输入函数的实参自动推断。

```
template< int lower, int upper, typename T>
T sum(const T arr[])
{
    T ret{ arr[lower] };
    for (auto i = lower + 1; i <= upper; i++)
        ret += arr[i];
    return ret;
}
int main() {
    int arr[]{ 1,2,3,4,5 };
    std::cout << sum< 0,4 >(arr) << '\t';
    std::cout << sum< 1,3 >(arr) << '\t';
    double arr2[]{ 1.2,2.2,3.4,4.7,5.8 };
    std::cout << sum< 1, 3 >(arr2) << '\t';
}
```

C++17中还允许用auto说明非类型模板参数，从而根据模板实例自动推断非类型模板参数的类型。例如下面的函数模板：

```
template< auto value > void f() { }
```

该函数模板的实例f<10>能自动推断非类型模板参数value的类型为int：

```
f<10>();                                    //从实例 f<10>自动推断 value 的类型为 int
```

下面为 constexpr 常量模板及其实例化代码：

```
template < typename T, T value > constexpr T TConstant = value;
constexpr auto const MySuperConst = TConstant < int, 100 >;
```

在 C++17 中可以用 auto 声明非类型模板参数，使得模板定义更加简单：

```
template < auto value > constexpr auto TConstant = value;
constexpr auto const MySuperConst = TConstant < 100 >;
```

而不需要在模板中显式地声明模板类型参数 T。

10.1.7 模板模板参数

有时候，实例化模板时，传递的不是一个数据类型或者具体值，而是另外的一个模板，即模板参数本身也是一个模板，这种模板参数称为**模板模板参数**。例如：

```
template < template < class > class X, class A >
void f(const X < A > &value) {
       / * ... * /
}
```

函数模板 f 的第 1 个模板参数 X 本身也是一个**类模板**(关于类模板，下面会介绍)。

在实例化函数模板 f()时传递给模板形参 X 的实参必须是一个模板。

10.1.8 模板参数的默认值

定义函数模板时，可以给类型模板参数、非类型模板参数、模板参数设置默认值。下面代码给类型模板参数 T、非类型模板参数 e 都设置了一个默认值：

```
template < typename T = int, int e = 2 >
T power(const T x) {
    T ret{ x};
    for (auto i = 1; i < e; i++)
        ret *= x;
    return ret;
}

int main() {
    std::cout << power(3)<<'\t';
    std::cout << power(3.5) << '\t';
    std::cout << power < double, 3 >(3.5) << '\t';
}
```

执行程序，输出结果：

```
9        12.25    42.875
```

可以看出，第 1 个类型模板参数可以由函数的实参自动推断其类型，而第 2 个非类型模板参数如果没有指定，就是默认值(e=2)。和函数的默认参数必须在非默认参数后面不同，默认模板参数可以在非默认模板参数的前面，即任何位置。即下面的函数模板也是没问题的：

```
template < typename T = int, int e >
T power(const T x) {
    T ret{ x};
    for (auto i  =  1; i < e; i++)
        ret  *=  x;
    return ret;
}
```

再看一个例子：

```
template < typename T  =  int, int e  =  2 >
T fun() {
    T  ret{0};
    for (auto i  =  1; i < e; i++)
        ret  +=  3.14;
    return ret;
}

int main() {
    std::cout << fun() << '\t';
    std::cout << fun < double >() << '\t';
    //std::cout << fun < 3 >() << '\t';           //错：3 实例化 T
    std::cout << fun < double, 3 >() << '\t';
}
```

该代码中第 1 个调用 fun()的模板参数都是默认值，第 2 个调用 fun < double >()指定了类型模板参数 T 为 double 类型，调用 fun < 3 >()是错误的，因为不能用数值 3 去实例化类型模板参数 T，最后的 fun < double, 3 >指定了 2 个模板参数。

执行程序，输出结果：

```
3       3.14     6.28
```

10.1.9 可变模板参数

第 6 章的 std::initializer_list 可定义可变数目的形参，但传递给形参的可变数目的所有实参类型都必须相同，而通过定义**可变模板参数**(variadic templates)的模板，可以给可变模板参数对应的形参传递不同类型不同数目的实参。

在模板头的 typename 关键字后跟 3 个点(…)说明一个模板参数是可变模板参数，即 template < typename… T >，说明 T 是一个可变模板参数。

函数模板的函数参数里包含这个可变模板参数的函数形参，例如：

```
template < typename... Args >
void fun(Args... args) {/ * ... * /}
```

说明了 Args 是一个可变模板形参(Args 也称为**模板参数包**,因为该模板参数表示的是多个参数),而 args 是 Args 类型的函数形参。

用 range for 访问如下传递给 Args 的每个实参是错误的:

```
template < typename... Args >
void print(Args...args) {
    for (auto arg : args)
        std::cout << arg;
}
```

这是因为 range for 默认所有元素的类型都是一样的。为了能访问 Args 中打包的每个实参,可以用递归的方法将该函数模板写成递归函数模板形式,为此,实际要定义 2 个函数模板:一个是不再需要递归的基情形;另一个是递归形式。

即 print()函数模板应写成如下 2 个函数模板:

```
template < typename T >
void print(T end) {                          //基情形: 只有一个函数形参
    std::cout << t << '\t';
}
template < typename T,typename... Rest >
void print(T first, Rest...rest) {           //递归情形: 有多个函数形参
    std::cout << t << "\t";   print(rest...);
}
```

递归形式的函数模板将可变模板参数拆成 2 部分:第 1 个是一个普通的模板类型参数;第 2 个是一个可变模板参数。函数模板也类似拆分,其中的 rest...表示逗号隔开的可变参数。调用这个递归函数模板,就可以传递多个实参:

```
int main() {
    print("Li") ;                            //调用的是 print(T end)
    print(2,"Li");                           //调用的是 print(T first, Rest...rest)
    print(2, "Li",80.5);                     //调用的是 print(T first, Rest...rest)
}
```

其中,第 1 句直接调用的是基情形版本的函数 print(T end),而多于一个实参如最后的 print(2, "Li",80.5)调用的是递归版本的函数 print(T first,Rest...rest),执行如下:

```
std::cout << 2 << "\t"; print("Li",80.5);
```

print("Li",80.5)调用的是递归版本的函数 print(T first,Rest...rest),执行如下:

```
std::cout << "Li"<< "\t"; print(80.5);
```

print(80.5)调用的是基本形版本的函数 print(T end),执行如下:

```
std::cout << 80.5 << "\t";
```

当然,非递归版本的函数也可以不包含任何参数,如下面的 sum()函数模板用于对一组可变模板参数求和。

```
#include <iostream>
auto sum() {     return 0;}

template <typename T1, typename... T>
auto sum(T1 s, T... ts) {return s  +  sum(ts...);}

int main() {
    std::cout << sum(1, 2) << std::endl;
    std::cout << sum(1, 2,3,4,5) << std::endl;
}
```

下面介绍**折叠表达式**。

针对可变模板参数,通常需要写 2 个函数(基情形和递归情形,递归情形用来解包参数)。C++17 的折叠表达式(fold expressions)是一种新的用运算符解包可变参数的方法。即用一个运算符(如 op)对打包的可变参数包(pack)处理,其常见格式如下:

```
(pack op ...)                              //(1)
(... op pack)                              //(2)
(pack op ... op init)                      //(3)
(init op ... op pack)                      //(4)
```

版本(1)是右折叠,即展开成(a_1 op (a_2 op (a_3... (a_{N-1} op a_N))))形式,版本(2)是左折叠,展开成(((((a_1 op a_2) op a_3) ...) op a_N)形式。版本(3)、版本(4)类似于版本(1)和版本(2),只不过多了一个初始值。

上面的 2 个 sum()函数(模板)可以用折叠表达式写成一个函数,其中的运算符是+。

```
template <typename... T>
auto sum(T... s){
    return (...  +  s);
}
```

用基于逗号运算符(,)的折叠表达式可将前面的 print()函数模板写成如下形式:

```
template <typename ...Args>
void print(Args ...args) {
    ((std::cout << args << '\t'), ...) << "\n";
}
```

当然，函数体中的代码也可以用一个 Lambda 函数做更多的工作，关于 Lambda 函数将在第 12 章介绍。

```
template < typename ... Args >
void print(Args ... args){
    ([](const auto& x) { std::cout << x << "\t"; }(args), ...) ;
    std::cout << "\n";
}
```

当然，折叠表达式还可以借助于 std::forward 等来处理。限于篇幅，这里不再介绍。

10.1.10 constexpr if

constexpr 使得表达式可以在编译期间进行计算，从而避免了运行期间的计算开销。如果和 if 结合，则可以在编译时根据常量表达式条件而丢弃 if 语句的分支。例如：

```
if constexpr (cond)
    语句 1;                          //如果 cond 是 false,则丢弃该语句
else
    语句 2;                          //如果 cond 是 true,则丢弃该语句
```

if 和 constexpr 的结合使模板代码的编写更加简单。例如下面代码是计算 fibonacci（斐波那契）数列的 C++模板实现：

```
template < int  n >
constexpr int fibonacci() { return fibonacci < n - 1 >() + fibonacci < n - 2 >(); }
template <>
constexpr int fibonacci < 1 >() { return 1; }
template <>
constexpr int fibonacci < 0 >() { return 0; }
```

在 C++17 中，如果用 constexpr if 实现则只需要写一个类似普通递归函数的模板。

```
template < int n >
constexpr int fibonacci(){
    if constexpr (n >= 2)
        return fibonacci < n - 1 >() + fibonacci < n - 2 >();
    else
        return n;
}
```

10.2 类模板

10.2.1 标准库类模板 vector

和函数模板类似，**类模板**是用于产生实际类的蓝图（设计图或模具）。如同函数模板是

一个参数化的函数,类模板则是一个**参数化**的类类型。通过用**类型模板参数**将数据元素类型**泛型化**,类模板可以生成针对不同模板实参的具体类,如 C++标准库的 vector 类模板是一个模板化的顺序表(称为向量。注意,不是数学上的向量),可以用不同数据元素类型作为模板参数去实例化不同的 vector 类。如:

```
#include <iostream>
#include <vector>                       //vector 类模板的头文件
using namespace std;
int main(){
    vector<int> ivec;                   //编译器会自动生成数据元素是 int 类型的 vector(向
                                        //量或数组)类
    vector<double> dvec;                //编译器会自动生成数据元素是 double 类型的 vector
                                        //(向量或数组)类
    vector<int> ivec2  = {3,5,7,4,6};   //用统一初始化{ }对 ivec2 初始化.ivec2 的类型是
                                        //vector<int>
    ivec = ivec2;                       //ivec 和 ivec2 是同类型 vector<int>的变量(对象),
                                        //可以相互赋值
    dvec = ivec2;                       //错: dvec 和 ivec2 是不同类型的对象,不能相互赋值.

    for(auto e: ivec2)                  //range for 可遍历向量 ivec2 的每个元素
        cout << e <<'\n';
    cout << endl;

    ivec.push_back(10);                 //在 ivec 向量类的最后加入一个整数 10
    ivec.push_back(9);                  //在 ivec 向量类的最后加入一个整数 9
    ivec.push_back(8);                  //在 ivec 向量类的最后加入一个整数 8
    for(auto e: ivec)   cout << e <<'\n';
    cout << endl;

    for(int i = 0; i!= 5;i++)
        dvec.push_back(2 * i + 1);
    for(int i = 0 ;i!= dvec.size();i++) //dvec.size()返回实际元素的个数
        cout << dvec[i]<<'\n';          //dvec[1]通过下标 i 访问其数据元素
    cout << endl;
}
```

上述代码中,分别通过**显式实例化**的方式,实例化了 2 个不同的 vector 类,即 **vector<int>**和 **vector<double>**。不同 vector 类是不同的数据类型,不能相互赋值。另外,push_back()用来在 vector 对象的最后添加一个新元素。

假如定义了一个表示学生的类 student:

```
#include <string>
using std::string;
class student{
    string name_;
    double score_;
public:
    student(string n,double s):name_(n),score_(s){}
```

```
    string name(){return name_;}
    double score(){return score_;}
    void set_name(string n) {name_ = n;}
    void set_score(double s){ score_ = s;}
};
```

同样可以直接在程序中实例化一个数据元素类型是 student 的 vector：

```
#incldue <iostream>
using namespace std;
int main(){
    vector<student> students;           //从 vector 实例化一个类：vector<student>
    student stu;
    cout <<"输入学生信息：name, score\n";
    while(cin>> stu.name){
        if(cin>> score&&score>=0)
            students.push_back(stu);
        else break;
        cout <<"输入学生信息：name, score\n";
    }
    cout <<"输出所有学生的信息\n";
    for(auto e: students)
        cout << e.name()<<'\t'<< e.score()<< endl;

    return 0;
}
```

vector 类模板是表示存储一组同类型数据元素的顺序表的模板，通过在 vector 模板后面用<>指定数据元素的类型就产生了一个具体的类，称为**类模板的实例化**。

类模板本身如 vector 不是一个类，类模板的实例如 vector<student>才是一个类，该类的完整名字是 vector<student>，而不是 vector。

从类模板可以实例化任意多个类，类模板代码只需要编写一次，编译器会通过实例化生成一个具体类的全部代码。和函数模板一样，这种强大的泛型技术大大提高了编程的效率，不需要为每种数据类型重复编写相同的代码。

vector 类模板实例化产生的类相当于 C++语言自带的数组，但它比 C++自带的数组具有更多优点，其中一个突出的优点是其大小可以动态变化，只要计算机内存足够，就能往里面放入任意多的数据元素，而 C++语言的数组是一种静态数组，即数组的大小必须在编写代码时就指定，一旦指定，程序运行过程中就不能改变。例如：

```
int main(){
    student students[100];              //100 个 student 的静态数组空间
    int num_student = 0;                //实际学生个数
    return 0;
}
```

这种静态数组在程序运行过程中不能修改，程序员在编写程序时，必须预估该程序将来

可以处理的学生的最多人数，但实际情况中学生数目是很难预估的，静态数组太大会造成空间浪费，太小又会造成空间不足。如同7.11节的Vector类一样，C++的vector（称为“**向量**”）则没有这个问题，并且可以表示任何类型的一个线性表，而7.11节的Vector类只能表示一个数据元素类型固定的线性表。

另外，和7.11节的Vector类一样，vector还有许多成员函数，如puch_back()、size()等，方便对vector实例化类的对象进行各种操作。

10.2.2 类模板 Vector

和函数模板一样，类模板由一个模板头开头，模板头由关键字template开头，后面用三角箭头(<>)说明其中有哪些模板参数。模板头的后面类似于普通的类定义，由关键字class开头，然后是类模板名，最后是类模板体。格式如下：

```
template < typename T >
class 类模板名{
  //类模板的定义
};
```

下面将改写前面的针对特定数据元素类型的类Vector，定义一个类似于std::vector的简化的类模板Vector。代码如下：

```
template < typename T                   //类型模板参数 T 用于泛化数据元素的类型
class Vector{
private:
    T * data{nullptr};                  //T * 类型指针指向数据元素类型，是 T 的动态内存块
                                        //(动态数组)
    size_t capacity{0};                 //动态空间的大小
    size_t n{0};                        //实际的数据元素个数
public:
    explicit Vector < T >(int cap = 5);
    ~Vector < T >();

    T& operator[](size_t index);                    //下标运算符
    const T& operator[](size_t index) const;        //const 版本的下标运算符

    Vector < T >(const Vector < T > & array);       //拷贝构造函数
    Vector < T > & operator = (const Vector < T > & rhs);  //赋值运算符

    bool push_back(const T &e);
    size_t size() const { return n; }               //返回实际数据元素的个数
};
```

类模板Vector中的代码类似于前面的类Vector的代码，只做了简单的修改。

(1) 首先，增加一个模板头，用类型模板参数T表示数据元素的类型。

(2) 代码中数据元素的类型都替换成了T。

(3) 类名从Vector变成了Vector<T>，因为Vector<T>才是完整的类型。

Vector 类模板仍然是 3 个私有成员变量：T * 类型指针变量 data 是存储所有数据元素的动态内存空间的起始地址；capacity 是这个内存块的大小(即可以存储多少数据元素)；n 是当前实际存放的数据元素个数，初始值为 0。

构造函数构造一个容量 capacity 是 5 的 Vector 类对象。如果将来存储空间已满，可以重新分配一块更大的内存块，让 data 指向这个更大的内存块并修改 capacity 的值。

除了构造函数和析构函数外，下标运算符 operator[]使得可以通过下标访问数据元素，而拷贝构造函数和赋值运算符使得可以复制(赋值)一个 Vector。push_back()方法每次向 Vector 里添加一个新的元素 e。size()返回当前实际元素个数。

该类模板目前只实现了一个成员函数模板 size()，下面逐一实现其他的成员函数模板。

10.2.3 定义类模板的成员函数

和普通的类一样，可以在类模板内部定义**成员函数模板**，例如 size()成员函数模板，也可以在类模板外部定义成员函数模板。因为成员函数模板是一个函数模板，而不是具体的函数，因此，如果在类模板定义的外部定义这些成员函数模板，就必须包含**模板头**(如这里的 template < typename T >)。

1. 构造函数模板 Constructor Templates

```
template < typename T >                         //类体外定义成员函数模板,必须包含模板头
Vector < T >::Vector(int cap) :data{ new T[cap] }, n{ 0 }
{
    if (data) { capacity = cap; }
}
```

在构造函数中，分配可以容纳 cap 个 T 类型元素的内存空间。

注意：

- 在 new T[cap]分配一个 T 类型的数组空间时，这个 T 类型必须有默认构造函数，如果 T 没有默认构造函数，创建这 cap 个 T 对象数组时就无法给 T 的构造函数提供初始化参数。
- 在类体外定义成员函数时，要写出完整的类名 Vector < T >而不是 Vector。

2. 析构函数模板 Destructor Templates

```
template < typename T >
Vector < T >::~Vector() {
    delete[] data;
}
```

析构函数很简单，就是用 **delete[]**释放构造函数中分配的 data 指向的内存块。

3. 拷贝构造函数模板

创建一个和已有 Vector 一样内容的 Vector，即新的 Vector 是已有 Vector 的一个复制(拷贝)。

```
template < typename T >
Vector < T >::Vector(const Vector& vec) : Vector{ vec.capacity }{
    if(!data)  return;
    n = vec.n
    for (size_t i{}; i < n; ++i)                          //拷贝每个数据元素
        data[i] = vec.data[i];
}
```

拷贝构造函数先根据被复制的 Vector 的 capacity 委托带 int 类型参数的构造函数分配足够大的存储空间，然后将已有对象的元素逐一复制到新对象的对应元素中，并设置新 Vector 的数据元素个数 n 等于 vec 的数据元素个数 vec.n。

4. 下标运算符[]模板

返回普通引用和 const 引用的下标运算符[]模板的内部代码是一样的，唯一的区别是：一个是 const 成员函数模板，返回的是 const 引用；另一个是普通成员函数模板，返回的是 non-const 引用。例如：

```
template < typename T >
T& Vector < T >::operator[](size_t index) {               //下标运算符
    if (index >= n) throw "下标非法";
    return data[index];
}

template < typename T >
const T& Vector < T >::operator[](size_t index) const {  //下标运算符 - const
    if (index >= n) throw "下标非法";
        return data[index];
}
```

const 和 non-const 的下标运算符函数模板具有一样的代码，实际编程中应尽量避免这种代码重复，因为将来如果想修改代码，如抛出标准库的异常 std::out_of_range，可能只修改了一个函数中的代码而忘记了修改另外一个函数的代码，会导致不正确的结果。避免重复代码(DRY)原则要求只能在一处编写代码，其他地方复用这个代码。保证这一处代码的正确性不但避免了单调的多处复制、粘贴、修改，也提高了程序的可维护性和可靠性。

因此，只要保留 const 版本的函数，non-const 对象可以调用这个 const 版本的函数，但反过来容易导致问题，因为对于 const Vector，不能直接调用 non-const 版本的成员函数。

那么 non-const 版本的下标运算符怎么调用 const 版本的下标运算符函数呢？能不能直接写成如下形式？

```
template < typename T >
T& Vector < T >::operator[](size_t index)                //下标运算符
{
    return (*this)[index];
}
```

答案是不行的。因为这是 non-const 版本的，因此 this 是 non-const 指针，*this 是 non-const 的引用，通过这个引用调用的下标运算符函数仍然是这个函数本身，因此，就产生了无限死循环。

正确的做法是：先将 this 或 *this 先通过强制类型转换（static_cast<>）为 const 类型的指针或引用，然后才能对这个 const 指针或引用调用 const 版本的下标运算符。

```
return static_cast<const Vector<T>&>(*this)[index];
```

将 *this 这个 **Vector<T>&** 引用强制类型转换为 **const Vector<T>&** 引用，并通过这个引用去调用 const 版本的下标运算符。但返回的是一个 const 对象的引用即 const T& 引用，而要实现的是 non-const 版本的下标运算符，返回的值应该是 T& 引用。即必须将上面的 const T& 转换为 T&，这时又需要用强制类型转换 **const_cast**<>去掉对象的 const 性，即 const_cast<>将 const 对象转换为 non-const 对象。因此，上述语句外面还要加上这个强制类型转换 **const_cast<T&>**：

```
return const_cast<T&> (static_cast<const Vector<T>&>(*this)[index]);
```

C++17 提供了一个辅助函数 std::as_const 可以将一个 non-const 对象强制转换为一个 const 对象。可以用它替代 static_cast<>，使代码更加简洁一些：

```
return const_cast<T&> (std::as_const(*this)[index]);
```

std::as_const 函数在头文件<utility>中，需要包含这个头文件。下面是完整的 non-const 下标运算符函数模板：

```
#include <utility>
template <typename T>
T& Vector<T>::operator[](size_t index) {                //下标运算符
    return const_cast<T&> (std::as_const(*this)[index]);
}
```

5. 赋值运算符模板

对于赋值运算，如果左操作数的空间容量大于或等于右操作数的数据元素个数，可以直接将右操作数的数据元素复制到左操作数的对应位置。但如果左操作数空间大小不足，需要重新分配足够大的空间，并释放原来的空间，然后再将右操作数的内容复制到这个更大的空间。

下面赋值运算符的实现代码直接采用后一种处理策略，即总是申请和右操作数一样大小的内存空间并释放原先的旧内存空间。

```
template <typename T>
Vector<T>& Vector<T>::operator=(const Vector& rhs)
{
    if (&rhs != this)                          //当右操作数不等于自己时，才赋值
```

```
    {
        T * temp = new T[rhs.capacity];      //申请和右操作数同样大小的空间
        if(temp){
            delete[] data;                   //释放原来的内存空间
            data = temp;
            //复制右操作数的内容
            n= rhs.n;
            capacity = rhs.capacity;
            for (size_t i{}; i < n; ++i)
                data[i] = rhs.data[i];
        }
    }
    return * this;                           //返回自身的引用
}
```

首先判断赋值语句的左、右操作数是不是同一个 Vector，如果不是同一个 Vector 才进行赋值。先申请一块和右操作数一样大的空间(temp= new T[rhs.capacity];)。如果分配新空间成功，则先释放自身原来的数据空间(delete[] data)，然后将 data 指向新的内存块(data = temp;)。接着修改空间容量 capacity 和数据元素数目变量 n，将右操作数的数据复制到左操作数这个新空间中的对应位置(for (size_t i{}; i < n; ++i) data[i] = rhs.data[i])。

该函数最后返回自身的引用，因为赋值运算符要求必须返回自身引用，从而赋值运算符可以连起来使用(如 vec2 = vec1=vec)。

赋值运算符函数内部是和拷贝构造函数基本同样的代码，按照 DRY 原则，应该避免重复代码。

是否可以在其中调用拷贝构造函数呢？

```
template < typename T >
Vector < T > & Vector < T >::operator = (const Vector& rhs)
{
    if (&rhs != this)                    //当右操作数不等于自己时,才赋值
    {
        Vector < T > ret(rhs);           //拷贝构造函数构造一个临时局部变量 ret
        delete[] data;                   //释放本来的内存
        data = ret.data;                 //data 指向临时局部变量的 data 内存
        n = ret.n;
        capacity = ret.capacity;
    }
    return * this;                       //返回自身的引用
}
```

该函数中定义了一个临时的局部变量 ret，就是右操作数 rhs 的副本，然后释放自身原来的空间，并将 ret 的值赋值给自身的 3 个成员变量。这看起来不错，然而存在一个致命的问题，左操作数和这个临时的 ret 的 data 指针值是一样的，即它们指向了同样的内存块，当 ret 退出它的作用域时，该变量被销毁，会调用析构函数，释放 ret.data 指向的这个内存块。左操作数的 data 指向的这个内存块已经不属于这个程序了，将来访问其中的内容时会引起

“非法内存访问”的错误，导致程序崩溃，如图 10-1 所示。

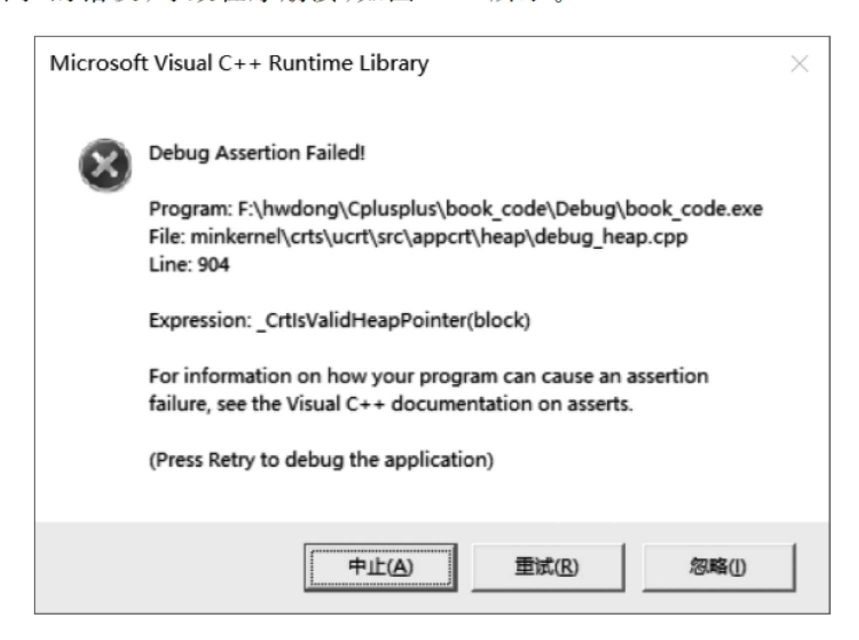


图 10-1　访问已经释放的内存或多次释放同一块内存都会导致程序崩溃

正确的方法是交换 2 个对象的 data 指针，当临时的 ret 被销毁时，销毁的是被赋值对象原先的旧的空间地址：

```
template < typename T >
Vector < T > & Vector < T >::operator = (const Vector& rhs)
{
    if (&rhs != this)                        //当右操作数不等于自己时,才赋值
    {
        Vector < T > ret{ rhs };
        //交换 ret.data 和 data 指针
        T * temp = ret.data;
        ret.data = data;
        data = temp;

        n = ret.n;
        capacity = ret.capacity;
    }
    return * this;                           //返回自身的引用
}
```

当然交换 2 个 data 指针变量的值可以使用标准库的函数 std::swap()。即代码也可写成：

```
template < typename T >
Vector < T > & Vector < T >::operator = (const Vector& rhs)
{
    if (&rhs != this)                        //当右操作数不等于自己时,才赋值
    {
```

```
        Vector<T> ret{ rhs };
        std::swap(data, ret.data);
        std::swap(n , ret.n);
        std::swap(capacity, ret.capacity);
    }
    return *this;                        //返回自身的引用
}
```

6. push_back()

push_back(const T& e)将一个新的元素 e 加到已有数据元素的最后面。但如果原有空间已满,需要分配更大的内存空间:

```
template <typename T>
bool Vector<T>::push_back(const T&e)
{
    if (capacity == n)                   //空间已满
    {
        T* temp= new T[2*capacity];      //分配 2 倍的内存空间
        if (!temp) return false;         //失败
        capacity *= 2;                   //设置空间容量
        //将数据从旧空间复制这个新空间
        for (size_t i{}; i<n; ++i)
            temp[i] = data[i];

        delete[] data;                   //释放本来的内存
        data = temp;
    }
    //将新数据 e 加到已有数据元素(下标 0,1, …,size-1)的最后面
    data[n] = e;   n++;
    return true;
}
```

在给一个 Vector 添加新元素时,push_back()应该检查是否有足够的空间存放新元素,当 n==capacity 时,说明空间已满,这时,应该分配一块更大的空间(T* temp= new T[2*capacity]; capacity *=2;),并将原来的数据复制到这个新空间中去(for (size_t i{}; i < n; ++i)temp[i] = data[i];)。然后释放原来的内存(delete[] data;),并使空间指针 data 指向这个新空间(data = temp;)。最后将新元素放到已有数据的最后(data[n] = e;),并更新数据元素的个数(n++;)。

当然,可以将上述增加空间容量的代码编写成一个单独的成员函数。

7. 测试 Vector 类模板

编写一段简单的代码,测试一下这个类似于 std::vector 的 Vector 类模板是否正常工作。

```
int main() {
    Vector<int> a;      //Vector<int>是 Vector 类模板的实例化类,其中的数据元素类型是 int
    for (auto i = 0; i<= 6; i++)
```

```
        a.push_back(2 * i + 1);
    a[3] = 30;
    for (auto i = 0; i != a.size(); i++)
        std::cout << a[i] << '\t';
    std::cout << '\n';

    Vector < int > b;
    b = a;
    for (auto i = 0; i != a.size(); i++)
        std::cout << b[i] << '\t';
    return 0;
}
```

Vector 的默认初始容量是 5，所以这里用 **push_back**()往 **Vector** < **int** >类对象 a 中添加了 7 个数据元素，并修改了下标是 3 的元素的值(a[3] = 30;)，然后用一个循环输出所有数据元素的值，接着将这个 a 复制给另外的 Vector < int >类对象 b，并输出 b 中的内容，看看是否一样。

该程序的结果是：

```
1    3    5    30    9    11    13
```

上述程序中，编译器遇到语句"Vector < int > a;"时，通过**显式实例化**的方法从类模板 Vector 自动生成了一个 Vector < int >的类。当再遇到"Vector < int > b;"时不会再生成同一个类，而是用刚才的类创建了该类的对象 b。传递不同的模板参数，就会生成不同的类。例如：

```
Vector < double > d;
```

会生成一个类 Vector < double >和类对象 d。Vector < int >和 Vector < double >是 2 个不同的类，这 2 个类的对象(指针或引用)之间是不能相互赋值的，因为它们是不同的数据类型。

10.2.4 类模板的模板参数推断

在 C++17 之前，类模板的参数必须显式指定，如上面的 Vector < int >，即不能从模板类对象的初始化式中自动推断模板参数类型，而 C++17 中类模板和函数模板一样，可以自动推断模板参数类型。在 C++17 中，下面代码是可以从初始化列表中推断出模板 Vector < T >的模板参数 T 的类型的：

```
Vector x{ 2., 4.3, 8.  };
```

但是

```
Vector y{ 2, 4.3, 8. };
```

则无法推断模板参数的类型，因为 2 是 int 类型，而其他的值是 double 类型。

再如，std::tuple 是一个可以容纳不同类型数值的模板类。

```
std::tuple t(1, 2, 3);                      //自动推断 3 个值的类型都是 int
std::tuple<int, int, int> t2(1, 2, 3);      //指定 3 个值的类型都是 int
std::tuple t3(1, 2., 3.f);      //自动推断 3 个值的类型是 int、double、float
```

可以用 get<i>函数模板获得一个 tuple 对象的某个下标是 i 的元素。

```
std::cout << typeid(std::get<0>(t3)).name()<<"\n";
std::cout << typeid(std::get<1>(t3)).name() << "\n";
std::cout << typeid(std::get<2>(t3)).name() << "\n";
```

10.2.5 类模板的专门化

和函数模板的专门化一样，也可以定义类模板的专门化。例如：

```
template <>
class Vector<const char *>
{
    //处理特定类型 const char * 的类 Vector...
};
```

如果一个类模板的所有模板参数都专门化为特定的类型，如上面的 Vector 只有一个类型模板参数 T，被专门化为特定类型 const char *，类模板 Vector 的这个专门化实际上是一个类而不是模板。当然这个类的完整的类名是 Vector<const char *>。这种所有模板参数都指定专门类型的类模板专门化称为**完全专门化**。还有一种为**部分专门化**，即只专门化部分模板参数，这样的部分专门化仍然是一个类模板。

对于下面的类模板：

```
template<typename T, int s = 10>
class X {
    //...
};
```

可以对其中的一个模板参数，如类型模板参数，指定其专门化为 const char *，得到下面的专门化类模板：

```
template<int s>
class X<const char *, s> {
    //…
};
```

注意：类模板中的模板参数 s 也必须出现在专门类模板名后的<>里，但不需要说明其类型。同样地，下面的专门化将类型模板参数专门化为 T * 指针类型。

```
template < typename T, int s >
class X < T * , s > {
    //…
};
```

另外需要注意的是，专门化模板的模板参数不能有默认值。

如果类模板有专门化，在类实例化时，编译器会根据传递的实际模板参数选择一个最佳匹配的类模板。例如：

```
X < const char * , 3 > x;
```

上面的 2 个专门化都能匹配，但是由于 const char * 是比更一般的 T * 指针更特殊的指针类型，因此，最佳匹配的是 template < int s > class X < const char * , s >类模板。

10.2.6 类模板的友元

和普通类的友元一样，可以用关键字 friend 在类模板里指定外部函数、类或其他模板为该类模板的友元。例如：

```
template < typename T >
class X{
    int a{ 0 };
    //...
    friend void fun();
    friend class A;
};

void fun() {
    X < int > x;
    X < double > y;
    std::cout << x.a << '\t' << y.a << '\n';
}
```

上述的函数 fun()和类 A 是 X 类模板的每个实例的友元，假如 X 有实例化的类 X < int >、X < double >，则 fun()和 A 的成员函数都可以访问类 X < int >或 X < double >的对象的私有成员。

```
int main() {
    fun();
}
```

执行程序，输出结果：

```
0          0
```

假如类模板里有一个友元函数模板，该友元函数模板使用的是类模板的模板参数。如：

```
template < typename T >
class Y{
    //...
    friend Y < T > * g(Y < T > & e);
};
```

则作为友元的函数模板 g 和类模板 Y 是同一个模板头，因此，g 的实例（如 g < int >）只能是同类型模板参数（int）的类模板 Y 的实例 Y < int >的友元，而不是其他类型模板参数（如 double）的 Y 实例（如 Y < double >）的友元。

例如：

```
template < typename T >
class Point{
public:
    T x, y;
    Point(const T x, const T y) :x(x), y(y) {}
    //...
    friend Point < T > * g(Point < T > & e);
};
template < typename T >
Point < T > * g(Point < T > & e) {
    Point < T > * p = new Point < T >(e);
    return p;
}
int main() {
    Point < int > e(3,4);
    auto p = g < int >(e);
    std::cout << p -> x << "\t" << p -> y << '\n';
    Point < double > x(3, 4);
    auto q = g < int >(x);                    //错：g < int >不是 Point < double >的友元
    std::cout << q -> x << "\t" << q -> y << '\n';
}
```

10.2.7 类模板 std::initializer_list <>

当用{}初始化列表去初始化其他变量或作为赋值运算符的右操作数时，编译器会自动创建一个 std::initializer_list <>实例化类的对象。例如：

```
auto il = { 10, 20, 30 };
```

对于右边的括号初始化列表{10，20，30}，编译器会根据其中值的类型推断出一个 std::initializer_list < int >实例化类，创建一个这个类的对象替换{10，20，30}，并返回这个对象的引用。

假如定义了一个类：

```
#include <iostream>
class Point {
    int x, y;
public:
    Point(int x0, int y0) :x{ x0 }, y{ y0 }{}
    void print() {
        std::cout << x << "," << y;
    }
};
```

创建 Point 类的对象时，则可以传递一个初始化列表{2,3}作为参数，初始化列表被转换为 std::initializer_list<int>的对象，编译器会寻找 std::initializer_list<int>作为参数的构造函数，如果没有找到，则会寻找参数个数一样多的构造函数，如这里使用 Point(int x0, int y0)，来构造 Point 对象。因此，下面代码中定义变量 p 是可以的，而定义变量 q 是不可以的。

```
int main() {
    Point p{ 2,5 };                    //ok,参数个数正好匹配
    Point q{ 2,5,3 };                  //错：编译器没有 3 个参数的构造函数
}
```

如果 Point 类中定义了 std::initializer_list<int>作为参数的构造函数，则就可以接收任意多个值的括号参数化列表对象。如：

```
class Point {
int x, y;
    public:
        Point(int x0, int y0) :x{ x0 }, y{ y0 }{}
        Point(std::initializer_list<int> list) {
            auto it = list.begin();     //begin()返回一个指示 list
                                        //第一个元素位置的迭代器
            x = *it++;                  //先将 it 指向的元素值 *it 赋值给 x,
                                        //然后 it++,使得 it 指向下一个元素
            y = *it++;
        }
        void print() {
            std::cout << x << "," << y;
        }
};
```

这时上述 main()函数的 p、q 定义调用的就是这个新的带 std::initializer_list<int>参数的构造函数。

前面定义的 Vector<>类模板只能通过 push_back()等函数向该 Vector<>类对象添加新的元素，如果为这个类模板定义带 std::initializer_list<>参数的构造函数，就可以用括号初始化列表直接创建包含一系列数据元素的 Vector<>类对象，即如下定义一个 Vector<>类对象：

```
Vector<int> vec{ 2,3,4,5,6 };
```

甚至可以定义数据元素是 Vector<>对象的 Vector<>对象，即：

```
Vector<Vector<int>> matrix{ {1,2,3,4},{5,6,7,8} };
```

下面是 Vector 的带 std::initializer_list<>参数的构造函数：

```
Vector(std::initializer_list<T> l) {
    data = new T[l.size()];
    if (!data) return;
    capacity = l.size(); n = l.size();
    auto i{ 0 };
    for (auto it = l.begin(); it != l.end(); it++,i++)
        data[i] = *it;
}
```

其中，l.begin()和 l.end()返回一种称为**迭代器**的对象（关于迭代器将在 13.3 节介绍），分别返回指向 l 的第一个元素的位置和最后一个元素的后一个位置。迭代器类似于指针，* 运算符作用于一个迭代器（即 *it），得到这个迭代器指向的那个元素。

执行下列程序：

```
int main() {
    Vector<Vector<int>> matrix{ {1,2,3,4},{5,6,7,8} };
    for (auto i = 0; i < matrix.size(); i++) {
        for (auto j = 0; j < matrix[i].size(); j++) {
            std::cout << matrix[i][j] << "  ";
        }
        std::cout << std::endl;
    }
}
```

输出结果：

```
1  2  3  4
5  6  7  8
```

10.3 实战：强化学习 Q-Learning 求解最佳路径

10.3.1 强化学习

机器学习方法主要分为监督式学习、非监督式学习和强化学习。在监督式学习中，每个样本都有一个正确的答案，通过许多答案已知的样本(x_i, y_i)学习一个函数 $y=f(x)$，其中的 x 是样本特征，而 y 就是正确的答案。例如前面的线性回归预测房屋价格就是一个典型

到环境的反馈奖励，这些奖励累加起来就构造了从初始状态 s_0 出发的总奖励：

$$R(s_0,a_0)+\gamma R(s_1,a_1)+\gamma^2 R(s_2,a_2)+\cdots$$

其中，折扣因子 γ 表示长期奖励的重要性，如果该值比较小，说明更关注短期的奖励，如果该值比较大，说明长期奖励也很重要，也就是用于平衡眼前利益和长远利益。如果 γ 比较小，说明今天奖励 1 元钱比将来的 1 元钱更重要。

因为从一个初始状态出发可能采取很多不同的动作序列，强化学习的目标是最大化如下的平均总奖励：

$$E_{s_0,a_0,s_1,a_1,\cdots}[R(s_0,a_0)+\gamma R(s_1,a_1)+\gamma^2 R(s_2,a_2)+\cdots]$$

即各种可能的状态动作序列的奖励的期望(平均值)。

Policy(策略)是一个状态到动作的函数，即 $\pi:S\rightarrow A$。策略规定了在某个状态 s 应采取哪个动作 a，即 $a=\pi(s)$。状态的价值函数 $V^\pi(s)$是在策略 π 下从状态 s 出发的平均总奖励：

$$V^\pi(s)=E_{s_0,a_0,s_1,a_1,\cdots}[R(s_0,a_0)+\gamma R(s_1,a_1)+\gamma^2 R(s_2,a_2)+\cdots \mid s_0=s,\pi]$$

即从初始状态 s_0 出发的各种可能的状态动作序列的奖励的期望(平均值)。

对于一个固定的策略 π，价值函数满足**贝尔曼方程**(Bellman equations)。

$$V^\pi(s)=R(s)+\gamma\sum_{s'\in S}P_{s,a(s')}V^\pi(s')$$

其中，$R(s)$是 s 状态下采用各种动作的期望直接奖励。即 s 状态的价值等于直接奖励和其后续状态价值的期望值之和。

可以定义一个状态的最优价值函数，即任意状态 s 的最优价值为所有策略下该状态价值的最大值：

$$V^*(s)=\max_\pi V^\pi(s)$$

可以证明，这个最优价值函数也满足贝尔曼方程：

$$V^*(s)=R(s)+\max_a\gamma\sum_{s'\in S}P_{s,a(s')}V^*(s')$$

如果定义如下的策略 $\pi^*:S\rightarrow A$：

$$\pi^*(s)=\underset{a\in A}{\operatorname{argmax}}\sum_{s'\in S}P_{s,a(s')}V^*(s')$$

即在一个状态 s 下，选择一个动作 a，使得在所有动作中执行该动作后的平均价值最大。对任意状态 s，按照这个 π^* 策略选择的动作 $a=\pi^*(s)$ 就能使状态 s 的价值取得最优值，也就是这个策略就是最优的策略。所以只要能求得最优价值函数，在任意状态 s，根据这个公式从一个状态的所有可能动作 $a\in A$ 中选择一个使 $\sum_{s'\in S}P_{s,a(s')}V^*(s')$ 最大值的动作 a，就是一个最优策略。

上述公式涉及概率，看起来复杂，初学者一时不能很好地理解也没有关系，这并不会影响对下面的强化学习算法 Q-Learning 的理解和实现。

10.3.2 Q-Learning

1. Q-Learning 算法原理

Q-Learning 是一种求 MDP 问题的最佳策略的方法。它不是计算一个状态的价值，而是通过计算(状态、动作)的价值来寻找最佳的策略。即用 $Q(s,a)$描述状态 s 下执行动作 a

的价值，也称为质量。对于最佳的策略 Q^*，同样满足贝尔曼方程：

$$Q^*(s,a) = R(s) + \gamma \max_{a'} Q^*(s',a' \mid s,a)$$

即 $Q^*(s,a)$ 等于平均直接奖励和其后续状态 s' 下的各个动作 a' 的最大值 $Q^*(s',a')$ 的一个折扣值，即 $\gamma \max_{a'} Q^*(s',a')$。

和求解方程的根的迭代法一样，为了求解这个最佳的 $Q(s,a)$，Q-Learning 通过一个简单的**时差更新**方法来迭代地逼近最佳的 $Q^*(s,a)$。

首先初始化所有的 $Q(s,a)$ 值为一个随机初始值（如 0），然后用下面的迭代公式不断更新 $Q(s,a)$ 的值：

$$Q^{new}(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$$

其中，r 是在状态 s 采取动作 a 的直接奖励，$r+\gamma \max_{a'} Q(s',a')$ 就是根据后续（状态、动作）价值 $Q(s',a')$ 计算出来的 $Q(s,a)$ 的目标值，用这个 $Q(s,a)$ 目标值和原来 $Q(s,a)$ 的误差 $(r+\gamma \max_{a'} Q(s',a')-Q(s,a))$ 来更新修改 $Q(s,a)$，使得 $Q(s,a)$ 尽量趋向这个更大的价值 $r+\gamma \max_{a'} Q(s',a')$。$\alpha$ 是学习率，表示学习的速度，其值大，表示更快地向值 $r+\gamma \max_{a'} Q(s',a')$ 靠拢；其值小，表示慢一点靠拢。可以将原来的 $Q(s,a)$ 和 $r+\gamma \max_{a'} Q(s',a')$ 看成实数轴上的 2 个点，γ 控制 $Q^{new}(s,a)$ 靠近哪个更多些。

Q-Learning 算法通常任意选取一个状态 s_0 出发，然后采用 ε 贪婪法选择一个动作 a_0，得到一个奖励 r_0 并过渡到一个新的状态 s_1，然后根据上述迭代公式更新 $Q(s_0,a_0)$，再从 s_1 出发，采用 ε 贪婪法选择一个动作 a_1，得到一个奖励 r_1 并过渡到一个新的状态 s_2，然后根据上述迭代公式更新 $Q(s_1,a_1)$，…，这个过程一直进行直到遇到最终状态，构成一个完整的探索序列：

$$< s_0, a_0, r_1, s_1, a_1, r_2, s_2, \cdots >$$

这个探索序列也称为一个 episode（**片段**）。通常需要经过很多次 episode，$Q(s,a)$ 才能收敛到最佳的 $Q^*(s,a)$。

Q-Learning 算法的过程如下：

```
初始化 Q(s,a) = 0
多次(如 200 次)episode:
    对每个 episode,选择一个出发状态 s,执行下面的循环:
        用 ε 贪婪法选择一个动作 a
        得到环境反馈的(r,s')
        如果 s'不是结束状态,则更新 Q(s,a),令 s = s'.否则,这次 episode 结束
```

2. ε 贪婪法

在一个状态 s，贪婪法总是在可选的动作 a 中选择一个 $Q(s,a)$ 最大的动作，以便最大化最终的总奖励。然而，刚开始时，任意 (s,a) 的 $Q(s,a)$ 并没有达到最大值，如果只采用贪婪法，容易被短期利益所迷惑而陷入局部最大，而不能找到全局最佳的策略。

ε 贪婪法思想是对贪婪法做一点修正，大概率情况下选取当前 $Q(s,a)$ 最大值的动作 a，但也会以较小的概率去随机选择一个动作，从而探索未知的路线。如设置 $\varepsilon=0.1$ 表示以 0.1 的概率从所有可能动作中任意选择一个可行动作，以 0.9 的概率采取最佳动作。ε 贪婪

法可以在利用已有知识和探索未知领域之间取得一个平衡。这个 ε 和学习率 α 都需要根据实际问题选择一个合适的值，而且经常会采用自适应的方法，在迭代过程中不断调整 ε 和学习率 α。开始时 ε 值比较大，以便探索更多可能性，迭代过程中，其值越来越小，以便更好地收敛到最佳值。为简单起见，下面的程序采用固定的 ε 值。

10.3.3 Q-Learning 的 C++实现

Q-Learning 算法是通过迭代的方法计算每个状态 s 下的每个动作 a 的状态动作价值 $Q(s,a)$。在某状态 s 执行某动作 a 后会得到环境的反馈信息：直接奖励、过渡到的下一个状态以及是否到达了终止状态。

下面的类模板 QT 表示在一个状态下执行一个动作 action（动作名 action_name）的 $Q(s,a)$值 Q_value、获得的直接奖励 reward、过渡到的下一个状态 next_state，即 QT 不但记录了执行动作的 $Q(s,a)$值，还包含了环境的反馈信息。类型模板参数 T 表示 Q 值的类型（如 double）。

```
template < typename T >
class QT {
public:
    std::string action_name;                          //动作名
    T Q_value;                                        //Q(s,a)的值
    int next_state;                                   //转移到的状态
    T reward;                                         //直接回报
    QT(std::string a = " ", const int ns = 0, T r = 0, T qvalue = 0) :
        action_name{ a }, reward{ r }, next_state{ ns }, Q_value{ qvalue } {}
};
```

一个状态下的所有动作的 QT <>可以用一个线性表 Vector < QT < >>表示，称为这个状态的 QT 表。可以用 using 给这个线性表类型起一个简短的类型别名：

```
using State_QT_Table = Vector < QT < T >>;            //一个状态的 QT 表
```

所有状态的 State_QT_Table 又可以用一个线性表 Vector < State_QT_Table >表示，同样可以用 using 给这个线性表起一个简短的名字：

```
using QT_Table = Vector < State_QT_Table >;           //所有状态的 QT 表
```

类模板 QLearning 存储所有状态的 QT 表，并实现了上述的 Q—Learning 算法。代码如下：

```
template < typename T >
class QLearning {
    using State_QT_Table = Vector < QT < T >>;        //一个状态的 QT 表类型
    using QT_Table = Vector < State_QT_Table >;       //所有状态的 QT 表类型
    QT_Table QT_table;                                //所有状态的 QT 表
public:
    QLearning();
    QLearning(const Vector < Vector < int >> &maze);
```

```
    void Q_Learning(const int MAX_EPISODES = 15,
        const T EPSILON = 0.2, const T ALPHA = 0.1, const T GAMMA = 0.9);

    int random_start_state();                   //随机选择出发状态
    //选择一个动作
    int choose_action(const int state, T EPSILON = 0.1);
    //从当前状态 state 和动作 action,获得环境的反馈
    //下一个状态 next_state、直接奖励 reward,next_state 是否终止状态
    bool get_env_feedback(const int state, const int action,
        int &next_state,T &reward);
    T max_Qvalue(const int s,int& action);      //状态 s 下的最大 Qvalue 及对应的 action

    bool is_terminal(const int state);          //是否终止状态
    void print();                               //打印 QT_table
    void print_Q();                             //只打印 Q 值
    Vector< int > shortest_path(const int state);
};
```

其中,QT_table 是整个系统的 Q 值和转移信息(直接奖励和下一个状态)表。构造函数 QLearning()用于初始化一个问题的 QT 表,即初始化状态转移信息和 Q 值。对于机器人寻找金币问题,这个构造函数可以如下编写:

```
template< typename T >
QLearning< T >::QLearning(){
    State_QT_Table s_QT_table;
    s_QT_table.push_back({ "s", 5, -1 });
    s_QT_table.push_back({ "e",1,0 });
    QT_table.push_back(s_QT_table);             //状态 0

    s_QT_table.clear();
    s_QT_table.push_back({ "w", 0, 0 });
    s_QT_table.push_back({ "e",2,0 });
    QT_table.push_back(s_QT_table);             //状态 1
    s_QT_table.clear();
    s_QT_table.push_back({ "s", 6, 1 });
    s_QT_table.push_back({ "w",1,0 });
    s_QT_table.push_back({ "e",3,0 });
    QT_table.push_back(s_QT_table);             //状态 2
    s_QT_table.clear();
    s_QT_table.push_back({ "w", 2, 0 });
    s_QT_table.push_back({ "e",4,0 });
    QT_table.push_back(s_QT_table);             //状态 3
    s_QT_table.clear();
    s_QT_table.push_back({ "w", 3, 0 });
    s_QT_table.push_back({ "s",7,-1 });
    QT_table.push_back(s_QT_table);             //状态 3
    s_QT_table.clear();
    QT_table.push_back(s_QT_table);             //状态 5 是终止状态,空的 QT 表
    QT_table.push_back(s_QT_table);             //状态 6 是终止状态,空的 QT 表
    QT_table.push_back(s_QT_table);             //状态 7 是终止状态,空的 QT 表
}
```

当然这个函数采用了硬编码方式将机器人寻金币问题的数据在构造函数里初始化,一种更方便的方法是将这些数据放在一个外部文件里,直接从文件读取数据初始化。

对于终止状态,插入的是一个空的 QT 表。因此,很容易根据一个状态的 QT 表是否为空,判断其是否是终止状态,成员函数 is_terminal()就是用于判断一个状态 state 是否是终止状态:

```
template<typename T>
bool  QLearning<T>::is_terminal(const int state) {
      return QT_table[state].size() == 0;
}
```

Q_Learning()函数是 Q-Learning 算法的实现。其参数 MAX_EPISODES 表示进行多少次 EPISODE,EPSILON 表示 ε 贪婪法的 ε 值,ALPHA 表示学习速率,而 GAMMA 表示未来奖励的折扣。代码如下:

```
template<typename T>
void QLearning<T>::QLearning(const int MAX_EPISODES,
    const T EPSILON , const T ALPHA , const T GAMMA){
    //循环的回合数
    for (auto episode = 0; episode != MAX_EPISODES; episode++){
        auto step_counter{ 0 };
        auto s = random_start_state();          //选择随机出发状态
        std::cout <<"step: "<< episode <<" start State: "<< s <<'\n';
        auto is_terminated{ false };
        print_Q();

        //update_env(S, episode, step_counter)
        while (!is_terminated) {                //循环直到一局游戏结束
            auto action = choose_action(s, EPSILON);  //根据状态选择动作
            int s_next;
            T R;
            is_terminated = get_env_feedback(s, action, s_next,R);  //获取环境的反馈
            auto  q_predict = QT_table[s][action].Q_value;
            auto q_target = R;                  //如果 s_next 是结束状态,
            if (!is_terminated) {          //如果 s_next 不是结束状态,就更新 q_target 值
                T max_qvalue;
                auto  max_action{ 0 };
                max_qvalue = max_Qvalue(s_next, max_action);
                q_target = R + GAMMA * max_qvalue;
            }
            QT_table[s][action].Q_value += ALPHA * (q_target - q_predict);  //更新 Q 值
            s = s_next;                         //进入下一状态
            //print_Q();
        }
    }
    return;
}
```

在 QLearning()函数中调用的辅助函数 random_start_state()为每个 Episode(探索片段)随机选择一个出发状态,choose_action(s, EPSILON)采用 ε 贪婪法为当前状态 s 选择一个动作 action,get_env_feedback(s, action, s_next,R)表示状态 s 下执行该动作 action 后环境反馈的直接奖励 R 和下一个状态 s_next,该方法的返回值表示 s_next 是否是终止状态,如果不是终止状态,则用 max_Qvalue(s_next, max_action)得到 s_next 状态下的最大 Q 值,并用它预测 Q(s,action)的目标价值 q_target,然后用该价值和 Q(s,action)原先的价值 q_predict 的差来更新 Q(s,action)的值:

```
QT_table[s][action].Q_value += ALPHA * (q_target - q_predict);  //更新 Q 值
```

下面是这些辅助函数的代码:

```
template<typename T>
T QLearning<T>::max_Qvalue(const int s,int& action) {
    auto s_QT_table = QT_table[s];
    T max = s_QT_table[0].Q_value;
    action = 0 ;
    for (auto i = 1; i != s_QT_table.size(); i++) {
        if (s_QT_table[i].Q_value > max) {
            max = s_QT_table[i].Q_value;
            action = i;
        }
    }
    return max;
}
template<typename T>
int QLearning<T>::choose_action(const int state, T EPSILON) {
    T rnd{ random_real(0.,1.)};
    if (rnd < EPSILON)
        return  random_int(0, QT_table[state].size() - 1) ;
    else {
        auto  action{ 0 };
        max_Qvalue(state, action);
        return action;
    }
}

template<typename T>
int QLearning<T>::random_start_state() {
    int s{0};
    do{
        s = random_int(0, QT_table.size() - 1);
    } while (is_terminal(s));
    return s;
}
template<typename T>
bool QLearning<T>::get_env_feedback(const int state, const int action
    ,int &next_state, T &reward) {
```

```
    next_state = QT_table[state][action].next_state;
    reward = QT_table[state][action].reward;

    return is_terminal(next_state);
}
```

同时还定义了 2 个辅助的打印函数用于输出 QT 表的信息：

```
#include <iomanip>                                          //std::setw
template <typename T>
void QLearning<T>::print_Q() {
    for (auto i = 0; i != QT_table.size(); i++) {           //遍历每个状态
        if (QT_table[i].size() == 0)continue;
        std::cout << "state " << std::setw(3) << i << ":" << std::setw(3);
        for (auto j = 0; j != QT_table[i].size(); j++)  //遍历输出该状态的 QT 表
            std::cout <<"("<< QT_table[i][j].action_name <<","
            << QT_table[i][j].Q_value <<")\t";
        std::cout << std::endl;
    }
    std::cout << std::endl;
}
template <typename T>
void QLearning<T>::print() {
    for (auto i = 0; i != QT_table.size(); i++) {           //遍历每个状态
        if (QT_table[i].size() == 0)continue;
        std::cout << "state " << std::setw(3) << i << ":"<< std::setw(3);
        for (auto j = 0; j != QT_table[i].size(); j++)  //遍历输出该状态的 QT 表
            std::cout <<"("<< QT_table[i][j].action_name <<","
            << QT_table[i][j].next_state <<","
            << QT_table[i][j].reward << "," << QT_table[i][j].Q_value << ")\t";
        std::cout << std::endl;
    }
    std::cout << std::endl;
}
```

下面的 main()函数针对机器人寻金币问题，用 Q-Learning 算法求出所有“状态-动作”的价值函数 $Q(s,a)$：

```
int main() {
    QLearning<double> ql;           //定义一个 QLearning 实例化类对象
    ql.print();
    std::cout << "观察 QT 表如果没问题,请输入任何字符开始执行 Q-Learning 算法\n";
    char ch;
    std::cin >> ch;
    ql.Q_Learning();                //用 Q-Learning 算法求解 Q(s,a)值

    //输出从一个状态 s 出发到达终止状态的最佳路径
    auto s{ 0 };
    auto path = ql.shortest_path(s);
    std::cout << "shortest_path from " << s << std::endl;
```

```
    for (auto i = 0; i != path.size(); i++)
        std::cout << path[i] << '\t';
    std::cout << std::endl;
}
```

main()函数用 Q-Learning 算法求出价值函数 $Q(s,a)$后，就可以用 shortest_path(s)确定从任意状态 s 出发到达目标状态的最佳路径。最后，输出路径上的所有状态(当然也可以输出动作，读者可修改一下代码)。

shortest_path()函数的代码如下：

```
template<typename T>
Vector<int> QLearning<T>::shortest_path(const int state){
    Vector<int> path;
    auto s{ state };
    while (!is_terminal(s)) {
        path.push_back(s);
        auto  action{ 0 };
        max_Qvalue(s, action);
        s = QT_table[s][action].next_state;
    }
    path.push_back(s);
    return path;
}
```

shortest_path()从初始状态 s_0 出发，在 s_0 的所有可能的动作 a 中选择 $Q(s_0,a)$最大的那个动作 a_0 执行，然后到达下一个状态 s_1，在此状态的所有可能的动作 a 中再选择一个使 $Q(s_1,a)$最大的动作 a_1 执行，这样，可以一直到达目标位置，从而得到一个最佳路径。将经过的所有状态记录在一个 Vector 对象 path 中。

执行上述程序后，会在最后输出最终的 Q 值和从 $s=0$ 状态出发的最佳路径：

```
…
step: 14 start State: 3
state   0:  (s,-0.271)  (e,0)
state   1:  (w,0)       (e,0.104751)
state   2:  (s,0.686189)        (w,0)   (e,0.0114738)
state   3:  (w,0.122555)        (e,0)
state   4:  (w,0.002268)        (s,0)

shortest_path from 0
0       1       2       6
```

可以看到从初始状态 $s=0$ 出发，最佳路径是经过状态 1、2 到达目标状态 6。

这个 Q-Learning 实现不同于网上针对特定问题的特定实现，它是一个通用性的 Q-Learning 算法实现。对于网上的“无痛 Q-Learning”的房间问题、迷宫问题、QL 玩 FlappyBird 游戏等特定问题，只要修改构造函数里初始化数据的代码或重新定义新的构造函数就可以了。

例如，针对迷宫问题，可以重新定义一个构造函数，从文件或其他地方读取迷宫的数据。假设用 Vector 类模板定义表示迷宫的矩阵（二维数组）：

```
Vector<Vector<int>> maze{ {0, 0, 0, 0},
                          {0, -1, 0, 0},
                          {0, -1, -1, 0},
                          {0, 0, 0, 1} };
```

其中，0 表示可通，−1 表示墙，1 表示目的地，可以用一个新的构造函数根据这个矩阵初始化 QT_table：

```
//从输入参数是 Vector<Vector<int>>的二维矩阵构造迷宫问题的 QT 表
template<typename T>
QLearning<T>::QLearning(const Vector<Vector<int>> &maze) {
const auto m{ maze.size() }, n{ maze[0].size() };
    int s{ 0 };
    for (auto i = 0; i<m; i++) {
        for (auto j = 0; j<n; j++, s++) {     //状态 s 即(i,j)位置
            State_QT_Table s_QT_table;         //s 状态的 QT 表
            if (maze[i][j]!= 0){               //终止状态
                QT_table.push_back(s_QT_table); continue;
            }
            if (i>= 1) {                       //可向上运动
                auto s_next = s - n;
                s_QT_table.push_back({ "U", s_next,
                      static_cast<T>(maze[i-1][j]) });
            }
            if (i<m - 1) {                     //可向下运动
                auto s_next = s + n;
                s_QT_table.push_back({ "D", s_next,
                      static_cast<T>(maze[i+1][j]) });
            }
            if (j>= 1) {                       //可向左运动
                auto s_next = s - 1;
                s_QT_table.push_back({ "L", s_next,
                      static_cast<T>(maze[i][j-1]) });
            }
            if (j<n - 1) {                     //可向右运动
                auto s_next = s + 1;
                s_QT_table.push_back({ "R", s_next,
                      static_cast<T>(maze[i][j+1]) });
            }
            QT_table.push_back(s_QT_table);
        }
    }
}
```

然后将 main()函数中的 QLearning 对象替换成从 maze 构造的对象并设置 Episode 次

数为 100(注意足够的次数才能保证收敛):

```
QLearning<double> ql(maze);
ql.Q_Learning(100);
```

再执行 main()函数,输出结果:

```
…
step: 99 start State: 1
state   0:  (D,1.24568e-06)      (R,0.00168747)
state   1:  (D,-0.40951)         (L,0)    (R,0.0172721)
state   2:  (D,0.00742345)       (L,0)    (R,0.098172)
state   3:  (D,0.379251)         (L,0)
state   4:  (U,0.000111738)      (D,0)    (R,-0.989225)
state   6:  (U,0.00013765)       (D,-0.521703)   (L,-0.468559)   (R,0.121553)
state   7:  (U,0.0689871)        (D,0.681574)    (L,0.00803532)
state   8:  (U,1.08622e-05)      (D,1.91709e-08) (R,-0.40951)
state  11:  (U,0.0738184)        (D,0.947665)    (L,-0.19)
state  12:  (U,8.07135e-07)      (R,0)
state  13:  (U,-0.19)    (L,0)   (R,0)
state  14:  (U,-0.1)     (L,0)   (R,0)

shortest_path from 0
0       1       2       3       7       11      15
```

作为练习,读者可以将该程序用于其他的强化学习问题。

10.4 习题

1. 下列的函数模板 swap()用于对 2 个对象的值进行交换,请补充函数体代码并对不同类型的数据测试这个函数模板。

```
template<typename T>
void swap(T &a, T &b) {
      //补充你的代码
}
//对 int、double、string 类型测试 swap 是否正确
int main{
      //补充你的代码
}
```

2. 下面的函数可以判断一个字符串是否是一个 float 类型的实数,请将它修改为模板,以便可以用于 double 实数的判断。

```
#include<sstream>
using namespace std;
bool isFloat(string s) {
```

```
        istringstream iss(s);
        float dummy;
        iss >> noskipws >> dummy;
        return iss && iss.eof();
}
```

3. 将下列求最小值的函数转换为函数模板，然后对不同类型的数组测试你的求最小值的函数模板。

提示：T 转换为类型模板参数，而 n 转换为非类型模板参数。

```
int Min(T arr[], int n){
    if(n<=0) throw "invalid index!";
    int m = arr[0];
    for (int i = 1; i<n; i++)
        if (arr[i] < m)
            m = arr[i];

    return m;
}
```

测试代码：

```
int main(){
    int a[] = {3,5,1,27,13,9};
    double b[] = {2.3, 25.4, 7.8,11.1,39.23};
    cout <<"min of a is:"<< Min(a,6)<< endl;
    cout <<"min of b is:"<< Min(b,6)<< endl;
    vector<string> strs = {"hello","world","student","score","teacher","hi"};
    cout <<"min of atrs is:"<< Min(strs,6)<< endl;
}
```

4. 编写一个在一个数组中查找一个值的 find()函数模板，如找到返回下标，否则返回－1。

5. 请用任意一种排序算法(如冒泡排序算法、选择排序算法)编写一个 sort()函数模板，并用不同的数据类型(如 int、double、string)的一组数据测试这个函数模板。如何使函数模板能对用户定义类型如表示学生的 Student 类型的数组进行排序？

6. 将第 6 章的快速排序算法改写成函数模板，并对不同类型数据测试该函数模板。

```
template<typename T>
int partition(Vector<T> &a, const int start, const int end) {
//补充你的代码
}

template<typename T>
void quicksort(Vector<T> &a) {
//补充读者的代码
}
```

7. 对第6题的快速排序函数模型定义一个针对具有 Vector< const char * >类型的专门化函数模板,并测试是否正确。如对下列 main()中的 v 测试排序结果是否合理。

```
int main() {
    Vector< const char * > v;
    v.push_back("hello");
    v.push_back("world");
    v.push_back("scott");
    //…
}
```

8. 下面程序的输出是什么?

```
#include < iostream >
void f(){   std::cout << "1";}

template < typename T >
struct B{
    void f(){std::cout << "2";   }
};

template < typename T >
struct D : B< T >{
    void g(){   f();   }
};
int main(){   D< int > d;   d.g();}
```

9. 下面的 Array 类是一个表示固定大小的数组的类模板,请实现其中的成员函数,并测试你的实现是否正确。

```
#include < iostream >
using namespace std;

template < typename T >
class Array {
private:
    T * ptr;
    int size;
public:
    Array(T arr[], int s);
    void print();
};
```

10. 给 Vector 类模板添加更多的功能,如删除最后一个元素,在开头位置、中间某个位置插入、删除一个数据元素,查找是否存在某个值的数据元素等,并测试这些新的功能。

11. 将第7章的链表类转换为类模板,并对不同数据类型测试该类模板。

12. 定义一个矩阵类模板 Matrix<>,并用 Vector 类模板的实例化类对象作为其数据成员,表示一个矩阵,对这个矩阵类实现一个带 std::initializer_list<>参数的构造函数和表

示矩阵行列数及初始值的构造函数，实现拷贝构造函数、赋值运算符、下标运算符、函数调用运算符()以及+、-、* 运算符。

13. 请将下列代码的 4 个函数模板用 constexpr if 修改为一个函数模板。

```
#include <string>
struct Student{
    int id;
    std::string name;
    float score;
};
template <int i> auto& get(Student &s) { if(i<0||i>2) throw "error"; }
template <> auto& get<0>(Student &s) { return s.id; }
template <> auto& get<1>(Student &s) { return s.name; }
template <> auto& get<2>(Student &s) { return s.score; }
```

14. 对于下列代码：

```
template<typename T>
class Y{
    //...
    friend <typename T2> Y<T2>* h(Y<T2>& e);
};
```

给函数 h<int>传递一个 Y<double>类型参数 e，h 是否可以访问 e 的私有成员？请编写代码验证你的判断。

15. 网上有一篇 Painless Q-Learning 的文章介绍了一个机器人从一个建筑物的任意一个房间走出建筑物的探索问题。图 10-3 所示是建筑物的房间布局。

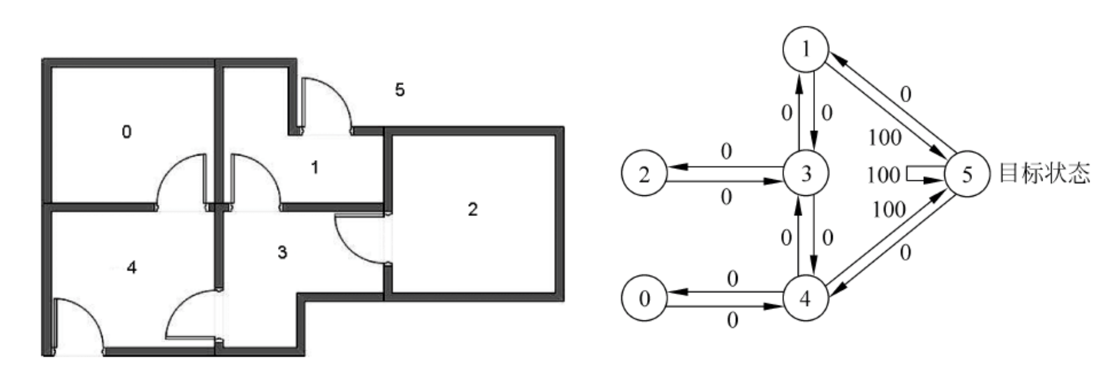


图 10-3 走出房间问题

将房间作为图的顶点，将 2 个房间之间的门作为边，可以构造如图 10-3 所示的一个图。边上的权值表示从一个门走向另外一个门的奖励。针对这个问题，请为 Q-Learning 编写一个构造函数，用于初始化这个问题的 QT 表，然后运行 main()函数，观察执行结果。注意，目标状态走向自身的这个边在程序中可以忽略。



移 动 语 义

11.1 左值和右值

11.1.1 左值和右值概述

C++的每个表达式可分为左值和右值(有时简写为 l-值和 r-值)。一个表达式不是左值就是右值。

左值具有可标识的内存地址并可以长时间存在;右值没有可识别内存地址,仅仅存储暂时的结果。左值的名称来源于历史上左值表达式,通常出现在赋值运算符的左边,而右值出现在右边。例如一个变量的表达式就是一个左值。左值、右值指的是表达式,而不是值。例如:

```
int var;
var = 3;
```

赋值语句的左操作数必须是一个左值,而变量 var 就是一个左值,因为它是一个具有可标识的内存地址。如果反过来:

```
3 = var ;
(var + 1) = 3;
```

则常量 3 和表达式(var+1)都不是左值(它们都是右值),这是因为它们是表达式的临时结果,没有可以标识的内存地址(即不能用取地址运算符得到它们的地址)。它们仅仅是计算过程中暂时保存在寄存器的临时值,所以给它们赋值是没有意义的(也无法给它们赋值)。

函数的返回结果表达式经常是一个右值,例如:

```
int foo() { return 3; }
```

如果对函数的结果赋值：

```
int main(){
    foo() = 3;
    return 0;
}
```

编译器会报告错误：

```
error C2106: "=": 左操作数必须为左值
```

并不是所有函数的返回结果都是右值，例如：

```
int g_var = 2;

int& f(){
    return g_var;
}

int main(){
    f() = 10;
    return 0;
}
```

函数 f()返回的是一个全局变量的引用，这个引用就是一个左值，可以给它赋值。如前面的类（如 8.1.2 节的 Point）的下标运算符函数通常有 2 个实现，其中返回的是引用（即左值），可以对返回的 non-const 引用赋值。

```
Point P;
P[1] = 3;
```

其中，P[1]是一个左值，因为它是 Point 类对象的成员变量 y 的引用，它是 Point 的 non-const 版本的下标运算符函数 Point::operator[]的返回值。

注意：并不是所有左值都是可以修改的。例如：

```
const int a = 10;          //a 是一个左值
a = 2;                     //但 a 不能被赋值(修改)
```

判断一个表达式是否是左值就是看它是否可以长期存在，即是否有一个地址，例如，可以用取地址运算符 & 帮助判断一个表达式是否是左值：

```
int main() {
    double a{}, b{ 1 }, c{ -2 };
    double *x = &(a + b);            //a + b 不是左值,无法取地址
    double *y = &a;                  //a 是左值
    double *z = &(std::abs(a * b));  //std::abs(a * b)不是左值
    double *u = &25;                 //常量 25 不是左值
}
```

& 作用于右值如 a+b,会产生编译错误(VS2017):

```
error C2102: '&' requires l - value
```

取地址运算符 & 作用于一个左值得到的表达式是一个右值,如上面的 &a 就是一个右值,因此,不能对它再取地址(&&a)了。

11.1.2 左值和右值的转换

左值可以转换为右值,例如上面的运算符 & 作用于一个左值得到的是一个右值。运算符实际是对右值进行计算的,因为参与运算的值都要放到处理器的临时寄存器中。运算符作用于左值,实际上会将左值隐式转换为右值(即移动到临时寄存器中)。

再如:

```
int a = 1, b = 2;
int c = a + b;
```

表达式 a+b 中左值 a 和 b 都会隐式转换为右值,然后参与运算,运算的结果也是在临时寄存器中,即也是右值。

右值不能转换为左值,但这并不意味着右值不能产生一个左值。例如:

```
int arr[] = { 1, 2 };
int* p = &arr[0];
*(p + 1) = 10;                    //ok: p + 1 是右值, 但 *(p + 1)是一个左值
```

其中,p+1 是右值,但 *(p+1)即 p+1 指向的对象是一个左值。

左值通常是可以修改的(但不是都如此),而右值是不可以修改的。C++11 之后,引入了**右值引用(rvalue references)**的概念,使得右值也是可以修改的。

11.1.3 左值引用和右值引用

前面的引用变量都是一种左值引用,即引用的是一个左值。C++11 中引入了**右值引用**的概念,即引用的也可以是一个右值。和左值引用一样,右值引用也是一个变量的别名,但它引用的是一个表达式的结果,尽管这个结果在一个临时内存里。绑定到右值引用会延长这种瞬态值的生命周期。只要右值引用在作用域内,就不会丢弃右值的内存。

通过 2 个 && 来定义一个右值引用,如:

```
int var{ 5 };
int& rcount{ var };              //rcount 是左值引用 lvalue reference,引用的是左值 var
int&& rtemp{ var + 3 };          //rtemp 是右值引用(rvalue reference),引用的是右值 var + 3
std::cout << rtemp << std::endl;  //输出 rtemp 引用的值
```

int&& rtemp{ var + 3 }定义了一个右值引用 rtemp 绑定到一个右值表达式 var + 3 的临时结果,在 rtemp 的作用域中,这个临时结果始终存在,因此,输出语句的输出结果

是 8。

总结：

- 可以绑定右值引用到一个右值，但不能绑定右值引用到一个左值。
- 普通引用不能绑定到一个右值，但 const 对象的引用可以绑定到一个右值。

例如：

```
int i = 42;
int &r = i;                //ok:  r 引用 i
int &&rr = i;              //错：不能绑定一个右值引用到一个左值上

int &r2 = i * 42;          //错：i * 42 是右值，普通引用 r2 不能绑定到一个右值
const int &r3 = i * 42;    //ok：可以绑定 const 对象的引用到右值
int &&rr2 = i * 42;        //ok：可以绑定右值引用到右值（临时变量）
```

右值引用主要用于 move 移动语义。

11.2 移动

11.2.1 复制和移动

对于一个类 X 和该类对象 y，定义变量"X x{y};"时会调用拷贝构造函数 X(const X&)，将 y 的内容复制到 x 中去。对于 X 的 2 个变量 x、y，执行 x=y 时，会调用赋值运算符函数，将 y 的内容复制到 x 中去。

这种复制是有一定开销的，对象如果占据内存越大，则复制开销越大。有时应该避免这种不必要的复制。先看下面的代码：

```
class X {
//...
public:
    X() = default;
    X(const X&) { std::cout << "拷贝构造函数!\n"; }
    X& operator = (const X&) {
        std::cout << "赋值拷贝函数!\n";
        return *this;
    }
};

X fun() {
    X t;
    //...
    return t;
}
int main() {
```

```
    X x;
    x = fun();
}
```

当执行 x=fun()时,会将 fun()返回值复制到临时变量,再将临时变量赋值给变量 x。一共执行了 2 次复制。执行上述程序,输出结果为:

```
拷贝构造函数!
赋值拷贝函数!
```

假如 X 类对象是一个占用内存比较大的对象,则 X 对象之间的复制开销就会比较大。

再看一个例子,对于前面的 Vector 类模板,在其中的拷贝构造函数和赋值运算符函数中各添加一条输出语句:

```
template < typename T >
Vector < T >::Vector(const Vector& vec) : Vector{ vec.capacity }
{
    cout << "拷贝构造函数: 复制了" << vec.n << "个元素\n";
    n = vec.n;
    for (size_t i{}; i < n; ++i)            //复制每个数据元素
        data[i] = vec.data[i];
}

template < typename T >
Vector < T > & Vector < T >::operator = (const Vector& rhs)
{
    if (&rhs != this)                       //当右操作数不等于自己时,才赋值
    {
        cout << "赋值运算符: \n";
        Vector < T > ret{ rhs };            //调用了拷贝构造函数
        std::swap(data, ret.data);
        n = ret.n_;
        capacity = ret.capacity;
    }
    return * this;                          //返回自身的引用
}
int main() {
    Vector < int > v1(20);
    Vector < int > v2;
    for (auto i = 0; i < 1000; i++)
        v1.push_back(2 * i + 1);
    cout << "左值的赋值...\n";
    v2 = v1;
}
```

程序中定义了 Vector 的 2 个对象 v1 和 v2，最后 v1 赋值给 v2。执行程序，输出结果：

```
左值的赋值...
赋值运算符:
拷贝构造函数: 复制了 1000 个元素
```

执行 v2 = v1 调用了赋值运算符函数，而在赋值运算符函数里，先执行了拷贝构造函数(Vector< T > ret{ rhs };)，然后和局部变量 ret 交换了数据指针 data。

由于赋值运算符里利用了一个临时变量 ret，该变量用 rhs 初始化，执行了拷贝构造函数。

再看一个右值的赋值：

```
Vector< int > f() {
    Vector< int > v(20);
    for (auto i = 0; i < 16; i++)
        v.push_back(2 * i + 1);
    return v;
}
int main() {
    Vector< int > v2;
    cout << "右值的赋值...\n";
    v2 = f();
}
```

执行程序，输出结果：

```
右值的赋值...
拷贝构造函数: 复制了 1000 个元素
赋值运算符
拷贝构造函数: 复制了 1000 个元素
```

函数 f()执行 return v 时，将结果保存到一个临时的变量即右值中，执行了一次拷贝构造函数，然后将这个右值赋值给 v2 时，又如上一段代码一样执行了赋值运算符函数的复制。

如果 Vector 实例化类的对象中数据元素很多(如机器学习等问题中样本个数经常是成千上万的)，假如有 10 000 个数据元素，而每个元素如果也是一个比较大的对象，如每个元素是一个 1000 个字符的字符串或者是一幅分辨率为 150 像素×150 像素的图像。这种对象之间的复制的开销就会很大。程序中如果多处地方有这种不必要的重复的复制，对程序性能影响很大。

C++11 开始引入的**移动语义**(**move semantics**)通过将一个对象的数据移动到另外一个对象中去，可避免复制的开销。所谓**移动**(**move**)是指将一个对象的内容移到另外一个对象中。如一个人将手中的某个东西(如一本书、一部手机、一辆汽车)转交给另外一个人，就是移动。而一个人复印别人的书使得 2 个人手中都有同一内容的各自一本书，就是**复制**。复制是一种昂贵的操作，有的情况下，应该尽量避免复制。

同样，如果将一个 Vector 对象的存储数据元素的内存块指针**移动**给另外一个 Vector

对象，就可以避免对象之间的不必要的复制。

为了能使 Vector 对象能够**移动**，需要给这个 Vector 添加**移动构造函数（move constructors）**和**移动赋值运算符函数（move assignment operators）**

11.2.2　移动构造函数

移动构造函数的参数是右值引用。在移动构造函数里，将右值引用对象的数据成员赋值给要构造对象的对应数据成员，然后将右值引用对象的数据成员设置为默认值，如指针设置为 nullptr，再如将下面代码中的 vec.data 指针设置为 nullptr。如果不这样做，则右值引用对象和新构造对象将共享同一个指针，导致同一块内存被（这 2 个对象的）析构函数多次释放。

```
//move constructor
template < typename T >
Vector < T >::Vector(Vector < T > && vec)
: n{ vec.n }, data{ vec.data }
{
    cout << "移动构造函数: 移动了 " << vec.size_ << "个元素\n";
    vec.data = nullptr;     //否则 vec.data 和正构造对象的 data 将共享同一块内存
}
```

即移动构造函数将右值引用 vec 的 vec.data 转移给要构造的 Vector 对象的 data，然后将 vec.data 设置为空指针（nullptr;）。这样既不需要分配新的存储空间，也不需要将数据一个个复制到新对象中。即没有任何数据复制的开销，提高了程序的效率。

11.2.3　移动赋值运算符函数

类似于移动拷贝构造函数，移动赋值运算符函数可以将一个右值引用对象的数据转移到另外一个左值对象中。

对于 Vector 类模板，代码如下：

```
//move assignment operator
template < typename T >
Vector < T > & Vector < T >::operator = (Vector < T > && rhs)
{
    cout << "移动赋值运算符: 移动了 " << rhs.size_ << "一个元素\n";
    if (this != &rhs)             //防止给自身赋值
    {
        delete[] data;            //delete[] 删除数据
        data = rhs.data;          //将右值引用对象的数据成员赋值给正构造对象的数据成员
        n = rhs.n;
        rhs.data = nullptr;       //右值引用对象的数据成员设置为默认值 nullptr
                                  //确保 rhs 不会调用 delete[]删除数据内存
    }
    return * this;                //return lhs
}
```

除了开始检查右值引用对象和要赋值的对象是否是同一个对象外，剩下的过程(将右操作数对象的数据成员赋值给左操作数对象的对应数据成员，然后将右操作数的数据成员设置为默认值，如指针设置为 nullptr)代码和移动构造函数基本是一样的。

再执行刚才的程序：

```
int main() {
    Vector< int > v2;
    cout << "右值的赋值...\n";
    v2 = f();
}
```

输出结果：

```
右值的赋值...
移动构造函数：移动了 1000 个元素
移动赋值运算符：移动了 1000 个元素
```

可以看到 f()局部变量 v 中的数据被移交给了 v2，避免了复制。

11.2.4 std::move

对于下面的代码：

```
Vector< std::string > str_vec;
Vector< std::string > string_vec;
for (auto i = 0; i < 10000; i++)
    string_vec.push_back("mljsdklfjsjfsyuqwhl");
str_vec = string_vec;
```

移动赋值运算符期待的是一个右值引用，而 string_vec 是一个左值，所以这里执行的是普通的赋值运算符，即复制操作。执行该赋值语句，输出：

```
赋值运算符：
拷贝构造函数：复制了 10000 个元素
```

通常情况下这是合理的，但如果语句“str_vec = string_vec;”是最后使用 string_vec 的地方，即后面不会再用到 string_vec，应该使用移动赋值运算符将 string_vec 移交给 str_vec 以避免复制的开销。但 string_vec 是左值，怎么办？C++标准库提供的 std::move()函数可以将一个左值转换为一个右值：

```
str_vec = std::move(string_vec);
```

此时调用的就是 Vector 的移动赋值运算符，将 string_vec 的数据移交给 str_vec，而 string_vec 自身的数据指针变为空。

执行该语句，输出：

```
移动赋值运算符：移动了 10000 个元素
```

std::move()并没有移动任何数据，仅仅将左值转换为一个右值。也就是说它做的工作就是类型转换。VS2017 中该函数的实现如下：

```
template<class _Ty>
_NODISCARD constexpr remove_reference_t<_Ty>&&
    move(_Ty&& _Arg) _NOEXCEPT
{   //forward _Arg as movable
    return (static_cast<remove_reference_t<_Ty>&&>(_Arg));
}
```

因此，std::move()就是强制类型转换 static_cast<>。

对于一个左值 lvalue，如果将 std::move(lvalue)传递给一个接受右值引用的函数，std::move(lvalue)就是一个右值引用；如果传递给一个接受左值(引用)的函数，它就是一个左值(引用)。

11.2.5 右值引用

右值引用的变量名是一个左值。例如：

```
Vector<std::string>&& ref{std::move(string_vec)};  //将 string_vec 转换为右值
                                                   //右值引用 ref 引用这个右值
Vector<std::string> str_vec2 = ref;                //ref 是一个左值，因此
                                                   //这是普通的赋值运算而不是移动赋值
```

执行该段代码，输出：

```
拷贝构造函数：拷贝了 10000 个元素
```

如果要执行移动赋值，则应该这样：

```
Vector<std::string> str_vec2 = std::move(ref);
```

即将右值引用 ref 先转换为右值，再执行移动拷贝构造函数。执行该语句，输出：

```
移动构造函数：移动了 10000 个元素
```

也就是说，右值引用引用的是一个右值，但其本身是一个左值，因为它有可识别的内存地址。

11.2.6 push_back()

下面是 push_back()函数：

```
template<typename T>
```

```
bool Vector<T>::push_back(const T&e)
{
    if (capacity == n)                          //空间已满
    {
        T* temp = new T[2 * capacity];          //分配 2 倍容量的内存
        if (!temp) return false;                //失败
        capacity *= 2;
        //将数据从旧空间复制到这个新空间
        for (size_t i{}; i<n; ++i)
            temp[i] = data[i];

        delete[] data;                          //释放本来的内存
        data = temp;
    }
    //将新数据 e 添加到已有数据元素的最后面
    data[n] = e;
    n++;
    return true;
}
```

该函数在空间已满时，需要将原有数据复制到新的空间中去(for 循环)。最后也将新的数据元素 e 通过复制方式复制到对应的数据元素(data[n] = e;)。如果数据元素的类型 T 占据内存较大，这种复制也是一个开销，可以用 std::move 避免复制：

```
for (size_t i{}; i<n; ++i)
    temp[i] = std::move(data[i]);
```

但对于 data[n] = e 则没法这样做，因为 e 是一个 const T&，不能转换为右值引用 T&&。解决办法是重载定义一个新的 push_back()成员函数，将函数的参数修改为右值引用：

```
template<typename T>
bool Vector<T>::push_back(T&& e){
    if (capacity == n) {                        //空间已满
        …
        for (size_t i{}; i<n; ++i)
              temp[i] = std::move (data[i]);    //移动
        …
    }
    //将新数据 e 添加到已有数据元素的最后面
    data[n] = std::move(e);                     //因为右值引用 e 是左值
                                                //所以要用 std::move 转换为右值
    …
}
```

11.3 习题

1. 结合例子解释什么叫左值引用和右值引用。它们之间有什么区别?
2. 编译下面代码,解释编译错误的原因。

```
int& foo(){
    return 2;
}
```

3. 什么类型的引用可以绑定到下面的初始化式子上?

```
int f();
int v[] = {1,2,3};
int ? r1 = f();
int ? r2 = vi[0];
int ? r3 = r1;
int ? r4 = vi[0] * f();
```

4. 说明下列代码的赋值语句两边的表达式哪些是左值,哪些是右值,代码是否存在错误,为什么?

```
int main(){
    int i, j, * p;
    i = 7;
    7 = i;
    j * 4 = 7;
    *p = i;
    ((i < 3) ? i : j) = 7;
    const int ci = 7;
    ci = 9;
}
```

5. 为第7、8章的String类定义移动构造函数和移动赋值运算符函数,并在这2个函数中添加一条打印语句,然后举例说明什么情况下这2个函数会被调用,并用代码验证这一点。

提示:如让一个函数返回String对象,将String对象通过std::move()转换为右值然后赋值,初始化另一个String对象。

6. 下列程序的输出是什么?

```
#include <iostream>
#include <utility>
int y(int &) {  std::cout <<"左值引用: ";   return 1; }
int y(int &&) { std::cout <<"右值引用: ";   return 2; }

template <class T> int f(T &&x) { return y(x); }
```

```
template <class T> int g(T &&x) { return y(std::move(x)); }
int main() {
    int i = 10;
    std::cout << y(i) << std::endl << y(20)<< std::endl;
    std::cout << f(i) << std::endl << f(20)<< std::endl;
    std::cout << g(i) << std::endl << g(20) << std::endl;
    return 0;
}
```

提示：函数模板中的 T && 不一定表示右值引用，它取决于用于实例化模板的类型。如果使用左值实例化，则它就是左值引用；如果使用右值实例化，则它就是右值引用。



函数指针、函数对象、Lambda表达式

C++和很多其他编程语言一样，提供了**头等函数**（**first class functions**），也就是说函数可以和变量一样使用，即可以将函数赋值给一个变量，作为另外函数的参数或返回值。C++通过**函数指针**、**函数对象**和 **Lambda 表达式**（也称**匿名函数**）提供头等函数功能。头等函数主要用作其他函数的参数。接受函数指针、函数对象或 Lambda 表达式的函数称为**高阶函数**。头等函数，特别是函数对象和 Lambda 表达式，在标准模板库中被广泛使用。

12.1 函数指针

12.1.1 函数类型和函数指针类型

一个程序中不但有数据还有代码，C++中不但可以定义指向数据的指针，还可以定义指向函数代码块的指针。如同通过变量的指针可以访问变量一样，也可以通过存储函数地址的指针去调用函数。

和指向变量的指针不仅仅是地址还包含了变量的数据类型一样，指向函数的指针除了包含函数地址外，还包含函数的参数类型和返回类型。和变量的指针一样，函数的指针也有类型，正如 int * 是一个指向 int 类型变量的指针类型。

对于下面这个函数：

```
double fun(const double * arr, const int n);
```

去掉函数规范中的函数名和参数名，就得到该函数的类型：double (const double * , const int)。

因此，该函数类型的指针的类型就是：double (*) (const double * , const int)，即在返回类型和参数列表之间添加了一对包含 * 的圆括号(*)。那么可以定义这个函数指针类型的变量如下：

```
double ( * pf) (const double * , const int)
```

即 pf 就是类型 double (const double * , const int)的指针变量,或者说 pf 指针变量的类型是:double (*) (const double * , const int)。

注意:包围 * pf 的一对圆括号不能缺少,否则含义就完全变了。假如将 pf 定义为 double * pf(const double * , const int),根据自右向左的阅读规则,pf 右边是一对带参数的圆括号,因此 pf 就是一个函数而不是函数指针,这个 pf 函数返回的是一个 int * 的指针。

在定义指向特定类型函数的指针变量时,可以给它一个初始值,如 nullptr:

```
double( * pf) (const double * , const int) = nullptr
```

也可以直接用一个这种函数类型的函数初始化它。假如有一个 double average(const double * , const int)的函数,那么可以用该函数初始化这个函数指针变量:

```
double ( * pf) (const double * , const int) = average;
```

或者也可以写成:

```
double ( * pf) (const double * , const int) = &average;
```

这两者都是一样的,即函数名 average 和前面添加了 & 运算符的 &average 都是函数 average 的地址。

```
double average(const double * arr,const int n) {
    return 0.;
}
#include <iostream>
int main() {
    double( * pf)(const double * , const int) = average;
    double( * pf2)(const double * , const int) = &average;
    std::cout << average <<'\t'<< &average << '\n';
    std::cout << * pf << '\t'<< * pf2 << '\n';
}
```

执行程序,输出结果:

```
008D1CDA        008D1CDA
008D1CDA        008D1CDA
```

即这 4 个地址值都是一样的。

假如 average()是一个求平均值的函数,可以通过函数的指针变量如 pf 调用这个函数,和通过函数名调用这个函数是一样的,即函数的指针相当于函数名(实际上它们都是同样的函数地址)。

```
double average(const double * arr,const int n) {
```

```
    auto ave = arr[0];
    for (auto i = 1; i != n; i++) ave += arr[i];
    ave /= n;
    return ave;
}
int main() {
    double(*pf)(const double *, const int) = average;
    double a[]{ 1,2,3,4,5 };
    std::cout << pf(a, 5) << '\t' << average(a, 5) << '\n';
}
```

执行程序，输出结果：

```
3       3
```

如果定义函数的指针变量用已有函数名初始化，则可以用 auto 关键字自动推断指针变量的类型：

```
auto pf = average;
```

12.1.2　给函数指针类型起别名

也可以用 using 给函数的指针类型起一个别名：

```
using AVE_PTR = double(*)(const double *, const int);
```

然后可以定义这个 AVE_PTR 类型的指针变量：

```
AVE_PTR pf = average;        //定义 AVE_PTR 类型的变量 pf
```

当然也可以用 typedef 给函数的指针类型起一个别名：

```
typedef double(*AVE_PTR)(const double *, const int);
AVE_PTR pf = average;        //定义 AVE_PTR 类型的变量 pf
```

using 和 typedef 都给类型 double(*)(const double *, const int)起了一个别名 AVE_PTR。typedef 的方法看起来复杂且不能用于函数模板的指针类型。因此，应该尽量用 using 而不是 typedef。

12.1.3　函数指针作为其他函数的参数

函数指针主要用作其他函数的参数，这个其他函数就是**高阶函数**，而作为参数的函数指针指向的函数就是所谓的**回调函数**。向高阶函数传递不同函数的指针，高阶函数中就调用不同的回调函数，从而执行不同的处理功能。

下面的函数模板 find_optimal 用于求数组 arr(大小为 n)的一个最佳值(如最大值、最小

值等）。

```
template<typename T>
T find_optimal(const T *arr,const int n, bool (*compare)(const T&,const T&) ) {
    T opt{ arr[0] };
    for (auto i = 1; i != n; i++)
        if (compare(arr[i], opt))  //arr[i]比 opt 更好
            opt = arr[i];
    return opt;
}
```

其中，第 3 个形参是一个比较 2 个值的函数指针 compare，compare 指向的函数就是回调函数，用于比较 T 类型的 2 个对象哪个更佳。而 find_optimal 就是高阶函数。调用函数 find_optimal()时，给形参 compare 传递不同的比较函数指针，就会得到按照不同比较函数求得的最佳值。

下面是 3 个比较 2 个对象的比较函数模板：

```
template<typename T>
bool less(const T& a, const T& b) { return a<b; }                              //小于

template<typename T>
bool greate(const T& a, const T& b) { return a>= b; }                          //大于或等于

template<typename T>
bool lessAbs(const T& a, const T& b) { return std::abs(a) < std::abs(b); }     //绝对值小优先
```

这 3 个函数模板具有同样的函数签名和返回值，可以给这些函数模板指针类型起一个别名 COMPARE_PTR：

```
template<typename T>
using COMPARE_PTR = bool (*)(const T&, const T&);
```

然后可以定义这个函数模板指针类型的变量，如：

```
COMPARE_PTR<double> pf;       //定义了一个 double 模板实参的函数指针变量 pf
```

COMPARE_PTR<double>就是一个模板参数 double 的函数指针类型，而 pf 是这种类型的一个变量。

可以给这个指针变量赋值不同的比较函数，并作为 find_optimal 的第 3 个形参，得到 find_optimal 函数模板的实例化函数：

```
int main() {
    double a[]{ 9, -3,2, -7};
    COMPARE_PTR<double> pf = less;                //auto pf = less<double>;
    std::cout << find_optimal(a, 4, pf) << '\t';
```

```
    pf = greate;
    std::cout << find_optimal(a, 4, pf) << '\t';

    pf = lessAbs;
    std::cout << find_optimal(a, 4, pf) << '\t';
}
```

执行程序，输出结果：

```
-7        9        2
```

上述代码中可以用 auto pf = less < double >代替 COMPARE_PTR < double > pf = less，使得代码更加简单，也不需要定义类型别名 COMPARE_PTR。

12.2 函数对象

函数对象（**function objects**，也称为**函子**）是一个实现了函数调用运算符()的类的对象，如同函数或函数指针一样，可以给函数对象传递参数，即可以像函数调用一样使用函数对象。例如：

```
class LessThan {
public:
    bool operator() (double a, double b) const{ return a < b; }
};

int main() {
    double x{ 3 }, y{ 4 };
    LessThan le;
    if ( le(x, y) ) std::cout << "x < y\n";
}
```

类 LessThan 定义了一个带 2 个参数 a、b 的函数调用运算符()，le 是该类的对象，而 **le**(**x**,**y**)实际调用的是 **le. operator**()(**x**, **y**)，这就是普通的函数调用。**le**(**x**, **y**)不过是 **le. operator**()(**x**,**y**)的简写而已。

上述代码的最后一句完全可以写成完整的函数调用形式：

```
if ( le.operator()(x, y) ) std::cout << "x < y\n";
```

因为 le(x,y)形式类似于函数调用，因此称 le 为**函数对象**，其实就是一个普通的类对象而已。定义了函数调用运算符的类对象称为**函数对象**。

使用函数对象的主要好处是因为它们是类对象，可以包含状态即数据成员变量。

函数对象作为一个普通的类对象，当然可以用作函数的参数或返回值。为了使上面的 find_optimal 不但能接受函数指针作为回调函数，还能接受函数对象来比较 2 个对象，可以添加一个模板参数 CompareT 表示对 2 个对象进行比较的函数对象或函数指针：

```
template < typename T,typename CompareT >
T find_optimal(const T * arr, const int n, CompareT compare){
    T opt{ arr[0] };
    for (auto i = 1; i != n; i++)
        if (compare(arr[i], opt))                    //arr[i]比 opt 更好
            opt = arr[i];
    return opt;
}
```

用作比较的类 LessThan 也修改成可以对任何类型对象比较的类模板：

```
template < typename T >
class LessThan {
public:
    bool operator() (T a, T b) const { return a < b; }
};
```

用 LessThan 类模板的实例化类对象去实例化 find_optimal 函数模板：

```
int main() {
    double a[]{9, - 3,2, - 7 };
    std::cout << find_optimal(a, 4, LessThan < double >()) << '\t';
}
```

当然 find_optimal 函数模板也能接收普通函数或函数指针：

```
template < typename T >
bool greate(const T& a, const T& b) { return a >= b; }
bool lessAbs(const double& a, const double& b) { return std::abs(a) < std::abs(b); }
int main() {
    double a[]{9, - 3,2, - 7 };
    std::cout << find_optimal(a, 4, LessThan < double >()) << '\t';
    std::cout << find_optimal(a, 4, greate < double >) << '\t';
    std::cout << find_optimal(a, 4, lessAbs) << '\t';
}
```

执行程序，输出结果：

```
-7      9      2
```

函数对象相对于普通函数的主要优点是函数对象作为一个对象，可以有状态(自身的数据)，而普通函数是独立封闭的、无状态的(当然静态变量也可以携带一定状态)。

例如，如何在不修改 find_optimal 内部代码的情况下，通过传入一个比较函数去查询和某个值 x 最接近的数组元素？

为此，需要能比较数据元素 a 和 b 哪个更接近 x，比较函数就应该是 3 个参数的 less(a, b, x)而不是 2 个参数的 less(a, b)。因为这时要修改这个函数的参数列表，增加一个接受用

户输入值的形参。如：

```
template < typename T >
bool nearest(const T& a, const T& b, const T& x) {
    return std::abs(a - x) <= std::abs(b - x);
}
```

也就是说这个函数 nearest 多了一个代表用户输入的值 x，而 find_optimal 的内部代码调用该函数的代码也要做相应的修改：

```
if (compare(arr[i], opt, x))
…
```

这样一来就破坏了原先的代码，反而使得原先求最小值或最大值的代码无法使用了。

函数对象可以完美地解决这个问题。只要在构造函数对象时，将用户输入值作为函数对象的数据成员(状态)保存在这个对象中就可以了。为此，可以定义如下的类：

```
template < typename T >
class Nearest {
    T x;
public:
    Nearest(T x) :x{ x } {}
    bool operator() (T a, T b) const {
        return std::abs(a - x) < std::abs(b - x); }
};
```

在程序中对于用户输入的一个值 x，创建一个函数对象 Nearest(x)，并传给 find_optimal 就能找到距离 x 最近的值：

```
std::cout << "\n 输入一个数值: \n";
double x;   std::cin >> x;
std::cout << find_optimal(a, 4, Nearest(x)) << '\t';
```

执行程序，输出结果：

```
输入一个数值:
1
2
```

下面介绍标准函数对象。

通过给 find_optimal 模板提供普通函数或函数对象，可以定制 find_optimal 的不同行为。C++标准库模板普遍采用这种方法使得程序员可以通过提供头等函数或函数对象来定制模板的行为。对于普通的函数，程序员很容易写出函数或函数对象；对于各种运算符，如比较或算术运算符，也可以写出相应的函数或函数对象，如前面的 LessThan <>、greate()，来模拟这些运算符。不过，C++标准库已经提供了这些运算符模板，可以直接拿来使用。例如 std::less <>模拟了<运算符，可以将它们直接作为高阶函数的参数。如：

```
std::cout << find_optimal(a, 4, std::less<>{}) << '\t';
std::cout << find_optimal(a, 4, std::greater<>{}) << '\t';
```

但必须包含这些标准函数对象的头文件<functional>。表 12-1 是这些运算符模板。

表 12-1 运算符模板

运算符类别	运算符模板
比较	less<>，greater<>，less_equal<>，greater_equal<>，equal_to<>，not_equal_to<> 即：<、>、<=、>=、==、!=
算术	plus<>，minus<>，multiplies<>，divides<>，modulus<>，negate<> 即：+、-、*、/、%、一元负号-
逻辑	logical_and<>，logical_or<>，logical_not<> 即：&&、||、~
位运算	bit_and<>，bit_or<>，bit_xor<>，bit_not<> 即：&、|、^、~

12.3 Lambda 表达式

12.3.1 定义和使用 Lambda 表达式

函数对象作为回调函数相对于函数或函数指针的优点是可以携带状态。然而编写函数对象对应的类的代码比较多，没有普通的函数简洁。Lambda 表达式提供了更好的解决方法，它兼具函数的简洁性，又可以像函数对象那样携带状态，并且还可以捕获其所在的包围环境中的数据。

Lambda 表达式的定义格式类似于函数，但不需要函数名，因此，也称为**匿名函数**。例如，12.2 节的 less 比较函数可以写成如下的 Lambda 表达式：

```
[](double x, double y) { return x < y; };
```

其中，()里面的是形参列表，{}里面的是函数代码。和普通函数定义不同的是其最左边有一对方括号[]，用来捕获该 Lambda 表达式的包围环境中的数据(变量)。

这个 Lambda 表达式没有一个函数名，那么如何使用它呢？一种方式是将它绑定到一个变量：

```
int main() {
    double x{ 3 }, y{ 4 };
    auto less =  [](double x, double y) { return x < y; };
    //auto less{ [](double x, double y) { return x < y; } };
    std::cout << less(x, y) << '\n';
}
```

上述代码给 Lambda 表达式一个变量名 less，即 less 就是这个匿名函数，可以给它传递

实参调用这个 Lambda 函数，如 less(x,y)。

另一种方式是直接调用 Lambda 表达式。例如：

```
#include <iostream>
int main() {
int sum{ 0 };
    for (int i{ 0 }; i != 5; i++)
        sum += [](int x) {return x * x; }(i);
    std::cout << sum << std::endl;
}
```

其中，黑体部分是 Lambda 表达式(函数)，该匿名函数带有一个形参 x。调用这个匿名函数时，通过(i)将实参 i 传递给形参 x，该函数返回 x 的平方值。执行程序，输出结果：

```
30
```

Lambda 表达式主要是作为高阶函数的回调函数。假如函数 accumulate()用来对一个数组 arr 中的每个元素用 fun()进行处理后再累加，例如：

```
int accumulate(int arr[],const int n,int ( * fun)(int) ) {
    int sum{ 0 };
    for (int i{ 0 }; i != n; i++)
        sum += fun(arr[i]);
    return sum;
}
```

通过给第 3 个参数(即函数指针参数)fun 传递不同的函数指针，就可以执行不同的处理并累加。

```
int Abs(int x) { return std::abs(x); }
int Square(int x) { return x * x; }

#include <iostream>
int main() {
int arr[]{ 3,5,7,9 };
    std::cout << accumulate(arr, 4, Abs) << '\n';
    std::cout << accumulate(arr, 4, Square) << '\n';
    std::cout << accumulate(arr, 4, [](int x) {return x * x * x; }) << std::endl;
}
```

该程序分别给 accumulate 的参数 fun 传递了 3 个不同的头等函数：Abs()、Square()和一个 Lambda 匿名函数。

执行程序，输出结果：

```
24
164
1224
```

再如,可以给前面的 find_optimal()的第 3 个参数传递一个 Lambda 表达式:

```
int main() {
    double a[]{ 7,1,5,9 };
    std::cout << find_optimal(a, 4, [](double x, double y) { return x < y; }) << '\t';
}
```

将对 2 个对象比较的 Lambda 表达式作为之前的 find_optimal 函数模板的第 3 个参数。

从 C++14 开始可以使用**通用 Lambda**(generic lambda)表达式,即 Lambda 的形参可以用 auto 关键字自动推断类型而无须给出显式的类型,因此,上面的 accument() 和 find_optimal()调用语句可以写成下面的形式:

```
accumulate(arr, 4, [](auto x) {return x * x * x; })
find_optimal(a, 4, [](auto x, auto y) { return x < y; })
```

12.3.2 捕获子句

如何类似于函数对象,将用户输入值传入给 Lambda 表达式? 办法就是将输入值传入 Lambda 表达式最左边的一对[]定义的**捕获子句**(capture clause)里。例如:

```
std::cout << "\n 输入一个数值: \n";
double v;   std::cin >> v;
std::cout << find_optimal(a, 4, [v](double x, double y) {
          return std::abs(x - v) < std::abs(y - v); }) << '\t';
```

将变量 v 传给 Lambda 表达式,执行程序,输出结果:

```
输入一个数值:
4
5
```

除了将单独的变量通过捕获子句传递给 Lambda 表达式外,还可以通过[=],即在捕获子句的一对方括号[]中放置一个=字符,表示 Lambda 表达式可以捕获其包围环境中的所有变量,这样就不需要将单独的输入值 v 传给 Lambda 表达式了。即:

```
int main() {
    double a[]{ 7,1,5,9 };

    std::cout << "\n 输入一个数值: \n";
    double v;  std::cin >> v;
    std::cout << find_optimal(a, 4, [ = ](double x, double y) {
        return std::abs(x - v) < std::abs(y - v); }) << '\t';
}
```

=捕获子句表示 Lambda 表达式的包围环境,即函数 main()的所有变量(如 v 和 a),都可以被 Lambda 表达式中的语句直接访问,虽然这里的 Lambda 表达式并没有使用变

量 a。

＝捕获子句使得包围环境中的变量是通过值传递的方式传递给 Lambda 函数，即这些变量的值被复制到 Lambda 表达式中。还可以用 & 子句即[&]将包围环境中的变量以引用方式传递给 Lambda 表达式。也就是说，Lambda 表达式中可以直接修改引用的包围环境中的变量。如：

```
int main() {
    double a[]{ 7,1,5,9 };
    int count = 0;
    std::cout << "输入一个数值: \n";
    double v;   std::cin >> v;
    std::cout << find_optimal(a, 4, [&](double x, double y) {
        a[count] -= v;
        count++;                                    //比较的次数累加
        return std::abs(x - v) < std::abs(y - v);
    });
    std::cout << "\t 比较的次数: " << count << '\n';
    std::cout << "修改了的 a 值: " ;
    for (auto e : a)
        std::cout << e << '\t';
}
```

通过[&]子句，在 Lambda 表达式中就可以直接修改包围环境的变量如 count、a。执行程序，输出结果：

```
输入一个数值:
4
5       比较的次数: 3
修改了的 a 值: 3   -3      1       9
```

用[&]捕获包围环境中的所有变量，可能会意外修改其中的变量而造成不易觉察的错误。可以用 & 捕获子句捕获一个单独的引用变量。如：

```
std::cout << find_optimal(a, 4, [&count](double x, double y) {
    //a[count] -= v;                              //错: 不能修改包围环境中的变量 a
    count++;                                      //比较的次数累加
    return std::abs(x - v) < std::abs(y - v);
    });
```

[**&count**]说明只捕获 count 作为引用变量，因此，在 Lambda 表达式中不能访问、修改其他的外围变量，如 a。也可以用捕获子句[**=，&count**]表示 count 作为引用变量，其他的外围变量作为值传递给 Lambda 表达式或者用[**&，count**]表示 count 作为值传递，其他的外围变量如 a 都作为引用传递。但捕获子句中不能同时包含＝和 &。捕获子句中单独的＝或 & 必须是第一项，如不能写成[count，&]或[count，＝]。

下面介绍捕获类的变量。

假如有一个类 X，其中有一个数据成员 value，还有一个成员函数 find_nearest()查询一

个数组 arr 中距离 value 最近的值，在该函数中调用了 find_optimal 来查找距离 value 的最近值。代码如下：

```
class X {
double value{ 0. };
public:
    X(double x) :value{ x } {}
    double find_nearest(const double arr[], const int n) {
        return find_optimal(arr, n, [value](double a, double b) {
                    return std::abs(a - value) < std::abs(b - value); });
    }
};
```

在调用 find_optimal()函数时，其中的 Lambda 表达式试图捕获该类的成员变量 value，但 value 属于一个类对象而不是属于整个类，这种方式编译器会报错：

```
1>… *.cpp(192): error C3480: "X::value": lambda 捕获变量必须来自封闭函数范围
1>… *.cpp (193): error C4573: "X::value"的用法要求编译器捕获"this",但当前默认捕获模式不
允许使用"this"
```

正确的方法应该是在 Lambda 的捕获子句中捕获调用 find_nearest()成员函数的那个对象的 this 指针，然后通过 this 指针访问 this—> value。代码如下：

```
return find_optimal(arr, n, [this](double a, double b) {
                    return std::abs(a - this->value) < std::abs(b - this->value); });
```

执行下面的程序：

```
int main() {
    double a[]{ 7,1,5,9 };
    std::cout << "输入一个数值: \n";
    double v;   std::cin >> v;
    X x(v);
    std::cout << x.find_nearest(a, 4) << '\n';
}
```

输出结果：

```
输入一个数值:
4
5
```

12.3.3 返回类型

Lambda 表达式可以从 return 语句推断出其返回类型，但有时无法自动推断，就需要明确指出其返回类型，这可以通过**尾置返回类型**的方式来说明，即在函数规范后面和函数体

之间用－>说明返回类型。例如：

```
[] compare(const int a ,const int b)  -> int {return a < b;}
```

12.3.4　Lambda 表达式的实质

Lambda 表达式的实质就是一个函数对象，编译器会将 Lambda 表达式转换为一个函数对象。例如，对于下面的 Lambda 表达式：

```
[](X &elem) {elem.op(); }
```

编译器会自动生成一个对应的类(类名是编译器生成)：

```
class_complieGeneratedName_ {
public:
    void operator() { X &elem }const {
        elem.op();
    }
};
```

其中，实现了函数调用运算符()，然后生成这个类的函数对象代替 Lambda 表达式。因此 Lambda 表达式就是一个函数对象。

如果 Lambda 表达式还捕获一个包围环境的变量，如：

```
[value](X &elem) {value++; elem.op(); }
```

编译器自动生成的类中会有一个对应的成员变量表示这个捕获的变量，假如 value 是 double 类型的，编译器生成的类可能是这样的：

```
class _complieGeneratedName_ {
    double value{};
public:
    void operator() { X &elem }const {
        value++;  elem.op();
    }
};
```

12.4　std::function

函数指针、函数对象、Lambda 表达式都可以作为可调用对象，即作为高阶函数的回调参数。然而它们是不同类型的对象，高阶函数要能同时接收这 3 种可调用对象，就必须将高阶函数定义成一个函数模板，并将可调用对象作为类型模板参数，如前面的 find_optimal 函

数模板那样。例如,下面的普通函数 fun()就不能同时接受函数指针和函数对象。

```
#include < iostream >
using namespace std;

int fun( int ( * pf)(int),int x ){
    return pf(x);
}

int add2(int x){   return x + 2; }

class AddOne{
public:
  int operator()(int x) {return x + 1;}
};

int main(){
    auto a{5};

    std::cout << fun(add2,a)<< std::endl;
    std::cout << fun(AddOne(),a)<< std::endl;    //error: cannot convert 'AddOne' to 'int ( * )(int)'
    std::cout << fun([](int x){return x * x;},a)<< std::endl;
    return 0;
}
```

再如,一个程序需要将这 3 种不同类型的可调用对象混合在一个数组里,即数组里的每个元素可能是这 3 种可调用对象之一,怎么办? 显然是不能直接这样做的,因为数组的数据元素类型必须是一样的。

为了解决上述问题,可以用函数模板 std::function 来统一包裹不同类型的可调用对象,即将不同类型的可调用对象包裹成 std::function 类型的函数对象。

std::function 是一个类模板,给它传递一个可调用对象的类型作为模板参数就可以实例化一个具体的 std::function 类,然后用这个 std::function 实例化类的对象来存储可调用对象。

对于形如"返回类型(参数列表)"的函数类型,如 int (const int x,const int y),可以用这个函数类型作为 std::function 的类型模板参数,得到一个实例化类:

```
std::function< int (const int x, cosnt int y)>
```

可以定义这个类的函数对象:

```
std::function< int (const int x, const int y)> compare;
```

可以用任何类型匹配 int (const int x,const int y)的可调用对象对 compare 赋值(或初始化):

```
int less(const int& a, const int& b) { return a < b?a:b; }
```

```
template<typename T>
int greate(const T& a, const T& b) { return a > b ? a : b;}

class  Less{
public:
    int operator()(const int& a, const int& b) { return a < b ? a : b; }
};

int main() {
    int a{ 3 }, b{ 8 };
    std::function< int(const int x, const int y)>  compare;
    compare = less;
    std::cout << compare(a, b) << '\t';
    compare = Less();
    std::cout << compare(a, b) << '\t';
    compare = greate<int>;
    std::cout << compare(a, b) << '\t';
    int v{ 3 };
    compare = [v](int a, int b) { auto x{std::abs(a - v)}, y{ std::abs(b - v)};
                                  return x < y?a:b; };
    std::cout << compare(a, b) << '\n';
}
```

执行程序，输出结果：

```
3       3       8       3
```

可以用一个数组统一管理这些不同的可调用对象，但这些可调用对象类型必须匹配 int(const int x，const int y)。如：

```
int main() {
    int a{ 3 }, b{ 8 };
    int v{ 4 };
    std::function< int(cosnt int x,const int y)> cmp_arr[]{
        less,Less(),greate<int>,
        [v](const int a, const int b) {
                auto x{std::abs(a - v)}, y{ std::abs(b - v)};
                    return x < y ? a : b; }
    };

    for(auto i = 0 ;i!= size(cmp_arr);i++)
        std::cout << cmp_arr[i](a, b) << '\t';
}
```

std::function< int(const int x，const int y)>类型的数组 cmp_arr[]中分别保存 4 个可调用对象：less、Less()、greate< int >和 Lambda 表达式([v](const int a，const int b))。

因此，标准库的函数模板 std::function 的实例化函数对象可存储同类型的函数、函数

对象、Lambda 表达式。

std::fcuntion 函数对象可作为函数的参数，用来指向不同类型(函数指针、函数对象、Lambda 表达式)的回调函数。如前面接受函数指针的函数 fun()的第一个形参可修改为 std::function<int(int)>，就可以接受任何不同类型的头等函数。代码如下：

```
#include <iostream>
#include <functional>
using namespace std;

int fun(std::function< int (int)> f, int x){
    return f(x);
}

int add2(int x){   return x + 2; }

class AddOne{
public:
  int operator()(int x) {return x + 1;}
};

int main(){
    auto a{5};

    std::cout << fun(add2,a)<< std::endl;
    std::cout << fun(AddOne(),a)<< std::endl;           //ok: 没有任何问题
    std::cout << fun([](int x){return x * x;},a)<< std::endl;
    return 0;
}
```

12.5 std::bind

std::bind 是一个函数适配器，它接受一个函数和一组实参，返回一个 std::function 函数对象。

假如有一个函数为 double square(double)，则可以将这个函数和一个实参 3.5 用 std::bind()绑定构造一个 std::function 函数对象：

```
auto f  =  std::bind(square,3.5);
```

也可以给 f 明确的 std::function 类型：

```
std::function<double ()> f =  std::bind(square, 3.5);
```

即 f 是一个返回值是 double 且不包含参数的 std::function 函数对象。作为 square()函数的包裹函数对象，f 已经包含了调用 square()函数的实参 3.5。

std::bind 和 std::function 都包含在头文件<functional>中，下面是完整的代码：

```
#include <functional>
double square(double x) { return x * x; }
int main() {
    auto f = std::bind(square, 3.5);
    std::cout << f();
}
```

执行程序，输出结果：

```
12.25
```

上面例子说明了 std::bind 可以将一个函数及其实参包裹在一个 std::function 函数对象中。如果不想将实参过早地包裹在这个函数对象中，而是将来调用这个函数对象时，传递某个实参，那么可以将这些实参用一个"占位符(placeholders)"代替。如下：

```
#include <functional>
#include <iostream>
using namespace std::placeholders;                    //引入名字空间 std::placeholders
int area(double pi, double r){
    return pi * r * r;
}

int main(){
    auto f = bind(area, 3.14, _1);
    std::cout << "3.14 * 2.5 * 2.5 " << " = " << f(2.5)<<'\n';
}
```

area()函数有 2 个参数，但在调用 std:: bind 时，只传递了第 1 个形参的实参，另外一个是以下画线开头的数字_1，这是一个占位符，其中的 1 表示它是第一个占位符。当调用这个函数对象 f 时，就应该传递一个实参给这个占位符，即对应 area()函数的第 2 个形参 r。

假如有一个函数：

```
void fun(int,const string &);
```

则可以进行如下绑定：

```
auto f1 = bind(fun,_1,_2);
auto f2 = bind(fun,_2,_1);
```

_1 和_2 表示函数 fun()的 2 个形参的占位符，则可以如下使用它们：

```
f1(3,"hello");
f2("hello",3);
```

12.6 习题

1. 定义 4 个对 2 个 double 类型的数进行加、减、乘、除的函数，然后将这 4 个函数的指针放入一个有 4 个元素的数组中，然后从键盘输入 2 个实数，并通过循环访问存储在数组中的函数指针去执行相应的函数。

2. 修改第 5 章的某个排序算法的函数，添加一个比较 2 个值大小的函数指针作为排序算法的形参。然后编写 3 个比较函数用于控制排序按照“从小到大”“从大到小”“绝对值从小到大”的方式排序。

3. 分别用函数对象和 Lambda 表达式代替第 2 题的函数指针，按 3 种不同方式对一组数据元素排序。

4. 下列程序计算 1～100 的整数之和，该程序有什么错误？请改正。

```
#include <iostream>
int main() {
    int sum = 0;
    for (int i = 1; i <= 100; i++)
        [sum](int x) {sum += x; }(i);
    std::cout << sum;
}
```

5. 补充下面程序的代码，用 Lambda 表达式对一个数组的相邻两项相加并将结果放到后一项中，假设数组有 n 个元素，即对每个下标 i(从 0 到 n−2)，执行 a[i+1] = a[i]+a[i+1]。

```
int main() {
    int arr[]{ 1,2,3,4,5,6,7,8 };
    for (int i = 0; i < 6; i++)
        //补充你的代码...
}
```

6. 为第 10 章的排序算法函数模板添加一个用于比较 2 个元素大小的类型模板参数，使得可以用普通函数或头等函数如函数对象、Lambda 表达式等控制排序的方式。

7. Lambda 表达式和普通函数的主要区别是能捕获其包围环境的变量，如下面 makeLambda()函数中的 Lambda 表达式可以捕获其包围环境，即 makeLambda()函数的局部变量 a。解释这段程序代码，并说明输出结果是什么。

```
#include <functional>
std::function<int(int)> makeLambda(int a) {
    return [a](auto b) { return a + b; };
}
#include <iostream>
int main() {
    auto add5 = makeLambda(5);
```

```
    auto add10 = makeLambda(10);
    auto f = add5(10) == add10(5);
    std::cout << add5(10) << '\t'<< add10(5)
        << '\t'<< f << std::endl;
}
```

8. 用函数对象代替 Lambda 表达式改写第 7 题中的代码。

9. 模仿下面的函数对象 functor，补充代码，给下面的头等函数数组 fs 再分别添加一个普通函数和 Lambda 表达式。

```
#include <functional>
#include <iostream>
class functor {
public:
    void operator()(const int i) const {
        std::cout << "i 的值是:" << i << std::endl;
    }
};
int main() {
    std::function<void(const int)> fs[3];
    fs[0] = functor();
    //补充代码: 添加普通函数或 Lambda 表达式到 fs 中
    //fs[1] = ?
    //fs[2] = ?
    int v;
    std::cin >> v;
    for (auto i{ 0 }; i != 3; i++)
        fs[i](v);
}
```

10. 定义一个具有函数声明 void fun(char，int，const string &)的函数，用 std::bind()从该函数和对应第 3 个形参的实参创建一个函数对象，其他 2 个形参对应 2 个占位符，然后测试该函数对象的使用。



C++标准库介绍

标准库是由 ISO C++标准制定的组件集，每个 C++实现都提供了以函数(模板)和类(模板)形式的 C++标准库实现。C++程序员应该尽可能使用这些经过严格测试的高效率的标准库中的函数和类库开发程序，既避免了重复编写代码，提高了开发效率，节省了开发成本，又可以保证程序质量和执行效率。

C++标准库中包含了很多内容：支持语言特征的如 range for、内存管理、类型检查的工具，支持并发计算的如线程、任务、同步锁等，支持输入输出的各种流库，存储数据集合的容器和迭代器，支持通用计算的算法，支持正则表达式，还包括 C 语言标准库和各种实用工具。

其中标准 ANSI C 库移植到 C++的库的头文件的名称都带有前缀 c 而不是后缀.h，例如，C 语言的< math.h >对应的< cmath >、C 语言的< string.h >对应的< cstring >、C 语言的< stdlib.h >对应的< cstdlib >等。但也有例外，如 C 语言的动态内存分配头文件< malloc.h >并没有对应的< cmalloc >头文件。

C++标准库有专门的字符串库，如 string 类(头文件< string >)，基于流的输入输出(简称 IO)类库(< iostream >、< iomanip >、< fstream >、< sstream >等)，用于内存管理的智能指针(头文件< memory >，如 shared_ptr <>、unique_ptr <>)等，各种容器(如< vector >、< list >、< set >、< map >等)，算法(< algorithm >)和迭代器(< iterator >)，正则表达式(< regex >)等。

标准库中很多都是以函数模板和类模板实现的，标准模板库(Standard Template Library，STL)是 C++标准库的核心。

本章仅介绍一些常用的标准库组件。更多 C++语言特征和这些标准库的内容可在下面 2 个网址查询。

(1) https://en.cppreference.com/w/cpp(C++参考手册)。

(2) http://www.cplusplus.com/。

13.1 输入输出流库

13.1.1 C++的 I/O 流库

1. I/O 流

C++语言本身并不提供输入输出(简称 I/O)功能,而是通过一组 C++类(模板)来处理输入输出。C++输入输出流库提供了格式化和非格式化的基于缓冲的数据输入输出功能。通过头文件如< istream >、< ostream >、< iostream >、< fstream >等提供各种具体的 I/O 流功能。

如图 13-1 所示,一个输出流 ostream 对象可以将不同类型的对象转换为一系列字符(字节)的流。

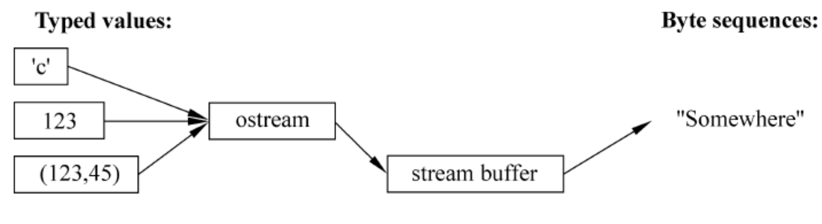


图 13-1 输出流 ostream

其中的流缓冲区(stream buffer)是 streambuf 对象,用于将一个 ostream 流对象映射到一个具体的设备。

同样地,如图 13-2 所示,输入流 istream 用于将一个字符(字节)流转换为不同类型的对象,流缓冲区负责 istream 对象和输入设备之间的映射。

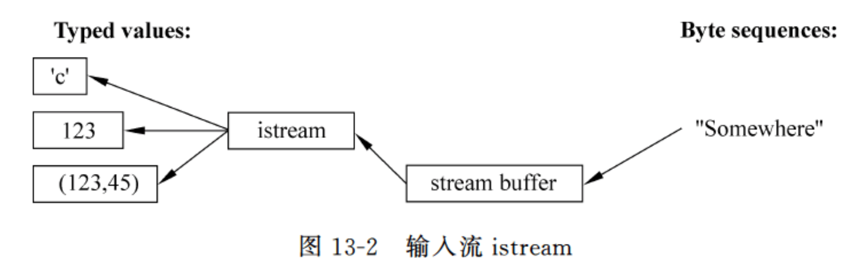


图 13-2 输入流 istream

I/O 流系统的关键组件如图 13-3 所示。

其中,实体箭头表示"派生于",虚线箭头表示"指向",<>表示模板。basic_iosream<>对象可以用于格式化输入输出,它派生自 basic_ios<>,basic_ios<>包含了本地化依赖的格式状态和流状态(**注**:本地化是指和本地语言相关的信息,如本地语言字符),它又派生自 ios_base,ios_base 描述了独立于本地化的格式状态。basic_ios<>中包含了指向本地化(locale)格式信息的指针和指向流缓冲区(basic_streambuf)的指针。basic_streambuf<>包含了指向具体读写设备和字符缓冲区(内存块)的变量。

通过将流对象绑定到不同的物理设备,就可以用统一的接口操作(如输入输出运算符>>和<<等)借助于流缓冲区对象在程序的各种类型对象和设备的字符(字节)流之间相互转

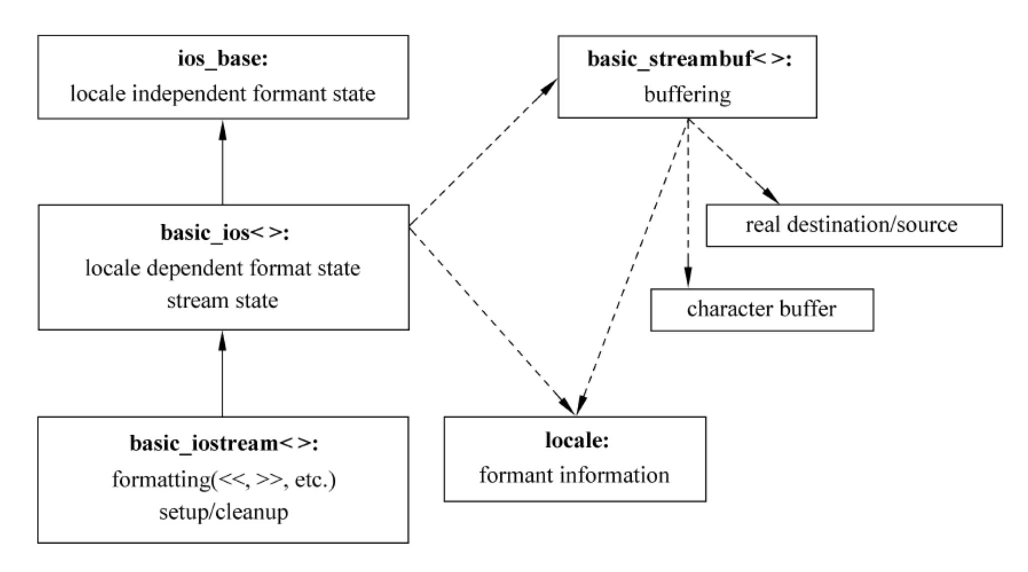


图 13-3 I/O 流系统的关键组件

换。不管具体的输入输出设备是键盘、控制台、内存、文件还是网络，不同类型对象的字符(字节)流的输入输出都是一样的，即程序员可以设备独立地进行 I/O 操作。

例如，下面程序向一个文件输出流 ofstream(定义在头文件 fstream 中)对象 outFile 用输出运算符<<输出不同类型的数据：

```
#include <fstream>
int main() {
    std::ofstream outFile("test.txt");
    outFile << 3.14 <<'\t'<<"hello"<<"\n";
}
```

上述代码定义 ofstream 类对象 outFile 时传递了文件名 text.txt，当创建对象 outFile 时就打开了这个文件，然后用输出运算符<<向 outFile 代表的文件 text.txt 输出实数、字符和字符串。这个输出过程和向标准输出流对象 cout 输出是完全一样的。

基于 I/O 流库的输出输入的一般步骤如下。

(1) 创建输入输出流对象。

(2) 连接输入输出设备。

(3) 执行输入输出(包括高层的格式化输入输出和底层的非格式化输入输出)。

(4) 断开输入输出设备。

(5) 释放输入输出流对象。

如果创建输入输出流对象时指定了关联的输入输出设备，则流对象的构造函数会将(1)和(2)同时完成，同样，在销毁一个关联输入输出设备的流对象时也会同时完成(4)和(5)。例如上面的"std::ofstream outFile("test.txt");"就同时完成了创建流对象和打开文件的工作，当该对象退出作用域时，自动调用 ofstream 类的析构函数，关闭文件和释放 outFile 占用的资源(如输出缓冲区)，即同时完成(4)和(5)的工作。

2. I/O 流层次结构

一个输入流(如 istream)对象可以连接到一个输入设备(如键盘)、网络端口、文件、字符

串，一个输出流(如 ostream)对象可以连接到一个输出设备(如控制台窗口和网络端口)、文件、字符串。针对不同输入输出设备、文件、字符串的输入输出流功能几乎都是以类模板编写，以支持不同的字符集(C++98/03 标准中的 char 和 wchar_t 及 C++11 标准引入的 char16_t、char32_t)，从这些类模板通过传递实际字符类型作为类型模板实参可得到实例化的类，如 istream、ostream、iostream、ofstream、ifstream、wistream、wostream、wiostream 等。

图 13-4 所示是 I/O 流的类层次结构。其中，虚线表示的是虚继承(指向的是虚基类)。

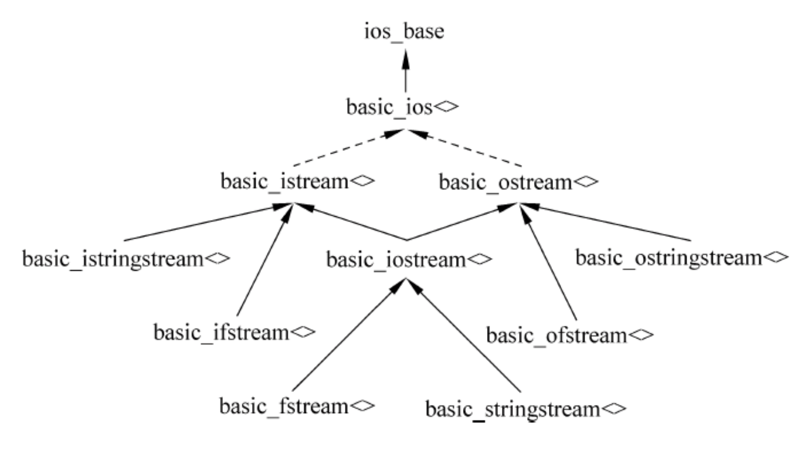


图 13-4 I/O 流的类层次结构

除类 ios_base 外，其他都是以 basic_开头的类模板，其中 basic_ios 是最关键的类，实现了输入输出流的绝大多数功能。大多数类模板都有 2 个类型模板参数。例如：

```
template < class charT, class traits = char_traits < charT > >
class basic_istream;
```

其中，charT 表示字符类型，例如 char 或 wchar_t。另一个类型模板参数 traits(默认是 char_traits < charT >)抽象了给定字符类型的基本字符和字符串操作的属性，例如字符集的排序次序，它使对这些字符的操作逻辑和存储分离。char_traits 类模板定义的操作集使得通用算法可用于几乎任何可能的字符或字符串类型。

使用特定的字符类型(如 char、wchar_t)实例化这些输入输出类模板，就得到一些针对特定字符类型的输入输出流类。如：

```
typedef basic_ios < char >            ios;
typedef basic_ios < wchar_t >         wios;
typedef basic_istream < char >        istream;
typedef basic_istream < wchar_t >     wistream;
typedef basic_ostream < char >        ostream;
typedef basic_ostream < wchar_t >     wostream;
typedef basic_iostream < char >       iostream;
typedef basic_iostream < wchar_t >    wiostream;
typedef basic_streambuf < char >      streambuf;
typedef basic_streambuf < wchar_t >   wstreambuf;
```

第 1 章的 cout 和 cin 就分别是 ostream 和 istream 类型的对象，都是针对 char 类型的

标准输出输入流对象。而 ofstream 也是针对 char 类型的文件输出流对象。

ios_base 和 ios：用于维护公共流属性的超类，例如格式标志、字段宽度、精度、区域设置等。超类 ios_base（不是模板类）维护独立于模板参数的流属性，而子类 ios（模板 basic_ios < char >的实例化）维护依赖于模板参数的流属性。

下面是一些针对 char 类型的实例化类。

istream（basic_istream < char >）和 ostream（basic_ostream < char >）：分别提供了输入和输出的公共接口。

iostream(basic_iostream < char >)：支持双向输入输出操作。而 istream 和 ostream 支持单向流操作。iostream 和 basic_iostream <>都定义在头文件< istream >中，该头文件和< ostream >头文件都包含在< iostream >头文件中。

fstream(basic_fstream < char >)、ofstream(basic_ofstream < char >)和 ifstream(basic_ifstream < char >)：用于文件输入、输出和双向输入输出，包含在头文件< fstream >中。

istringstream(basic_istringstream < char >)、ostringstream(basic_ostringstream < char >)和 stringstream(basic_stringstream < char >)：用于字符串缓冲区输入、输出和双向输入输出，包含在头文件< sstream >中

streambuf、filebuf 和 stringbuf：为流、文件流和字符串流提供内存缓冲区，以及用于访问和管理缓冲区的公共接口，包含在头文件< streambuf >中。

这些类可通过< **iostream** >、< **fstream** >(用于文件 I/O)和< **sstream** >(用于字符串 I/O)3 个头文件提供。此外，头文件< **iomanip** >提供了诸如 setw()、setprecision()、setfill()和 setbase()等用于格式化的操纵符。

< iostream >头文件中包含了头文件< ios >、< istream >、< ostream >和< streambuf >，并且定义了标准流对象 cin、cout、cerr 和 clog，它们分别对应于标准输入流（默认是键盘）、标准输出流、未缓冲的标准错误流和缓冲的标准日志流。cout、cerr 和 clog 默认对应的是控制台窗口。

3. 文本和二进制

所有文件可以分成 2 种：文本格式和二进制格式。虽然它们的内容都是以 0、1 串表示的，但它们的编码是完全不同的。文本文件是基于字符编码的文件，常见的编码有 ASCII 编码、Unicode 编码等。而二进制文件是以数值的计算机内部表示形式的二进制来表示的。例如，一个整数 123456 以文本表示就是用字符编码如 ASCII 码来表示该整数的每个字符 1、2、3、4、5、6，而二进制表示就是这个整数的计算机内部表示的二进制串（如 4 字节表示一个整数）。文本表示的数据可以直接以可读的字符形式显示，因此，打开一个文本文件，人们就能直接阅读其内容，而二进制表示是机器内部的表示，无法用记事本等文字处理程序阅读其内容。以文本文件存储数据通常需要更多的存储空间，如整数 123456 用 ASCII 表示需要 6 字节；而二进制文件通常需要的空间就相对较少，如整数 123456 只要 4 字节。因此，二进制文件比文本文件更小，可以提高读写的速度。

13.1.2 格式化输入输出

格式化输入输出是通过输入运算符(>>)和输出运算符(<<)进行的，并通过在< iomanip >

和< iostream >中的操纵符控制输入输出的格式化。

输入运算符(>>)和输出运算符(<<)都是以重载的函数模板的形式实现的。例如针对std::basic_ostream 的输出流运算符(<<)就有 11 个重载的版本，下列是其中的 2 个：

```
template < class CharT, class Traits >
basic_ostream < CharT, Traits > & operator <<( basic_ostream < CharT, Traits > & os,
                                               CharT ch );
template < class Traits >
basic_ostream < char, Traits > & operator <<( basic_ostream < char, Traits > & os,
                                              const signed char * s );
```

这些重载函数模板的第一个参数就是流对象的引用，返回值也是这个流对象的自引用。在此基础上，针对特定字符类型的实例化输入输出流类也提供了这些运算符的多个重载版本，例如针对 char 字符类的 ostream 类就提供了多达 17 个输出流运算符的重载版本。下面是其中的 3 个：

```
ostream& operator << (float val)
ostream& operator << (streambuf * sb);
ostream& operator << (ios_base& ( * pf)(ios_base&));
```

< iomanip >头文件中提供了 setw()、setprecision()、setbase()、setfill()等操作符控制输出项宽度、精度、进制和填充等。

setw(w)：指定输入输出项的最小宽度为 w 个字符。

setprecision(n)：浮点精度为 n，默认为 6。也可以使用成员函数如 cout. precision(n)设置浮点精度。

setbase(b)：按照 b 进制输出整数。

setfill(c)：用 c 填充空白，可以和对齐操作符一起使用。

例如：

```
#include < iostream >
#include < iomanip >
using namespace std;

int main(){
    cout << setw(12)<< 12345 <<' * '
        << setw(3)<<"hello"<< std::endl;
    cout << std::setbase(16)<< 100 << std::endl;
    cout << std::setbase(8)<< 100 << std::endl;
    cout << std::setbase(10)<< 100 << std::endl;

    cout << std::setfill ('x') << std::setw (10)
              << 100 << std::endl;
    double f  = 3.14159;
    cout << f << '\n'<< std::setprecision(3) << f << '\n';
    return 0;
}
```

执行程序,输出结果:

```
        12345 * hello
64
144
100
xxxxxxx100
3.14159
3.14
```

<iostream>中提供了 boolalpha/noboolalpha、left/right/internal、dec/hex/oct、flush/endl/unitbuf、fixed/scientific 等控制 bool 值的显示形式、左右对齐、进制、刷新缓冲区、定点格式及科学记数法等。

boolalpha:将 bool 值显示为字符串"true"或"false"。

noboolalpha:将 bool 值显示为 1 和 0。

dec:十进制。

hex:十六进制。

oct:八进制。

flush:刷新输出缓冲区,强制立即输出。

endl:插入换行符,然后刷新缓冲区,相当于先\n,然后再 flush。

left:左对齐,值的右边填充字符。

right:右对齐,值的左边填充字符。

internal:左对齐正负号,右对齐数字,在符号和值之间填充字符。

unitbuf:在每个输出之后刷新缓冲区。

fixed:用小数形式显示浮点数,固定显示小数点后的个数。

scientific:用科学记数法显示浮点数。

限于篇幅,不详细列举所有的操纵符,读者可查看以下网址的文档说明及例子代码。

(1) https://zh.cppreference.com/w/cpp/io/manip(中文)。

(2) https://en.cppreference.com/w/cpp/io/manip(英文)。

例如:

```
#include <iostream>
#include <sstream>
int main(){
    std::cout << " 0.01 的定点格式(fixed): " << std::fixed << 0.01 << '\n'
        << "0.01 的科学记数法(scientific): " << std::scientific << 0.01 << '\n'
        << "0.01 的十六进制(hexfloat): " << std::hexfloat << 0.01 << '\n'
        << "0.01 的默认格式(default): " << std::defaultfloat << 0.01 << '\n';
    double f = 3.1415926;
    std::cout << f <<'\n'<< std::setprecision(10)<< std::fixed << f <<'\n'
          << 3.14 <<'\n';
    std::istringstream("0x1P-1022") >> std::hexfloat >> f;
    std::cout << "将 0x1P-1022 解析为十六进制的结果是: " << f << '\n';
}
```

执行程序，输出结果：

```
0.01 的定点格式(fixed): 0.010000
0.01 的科学记数法(scientific): 1.000000e-02
0.01 的十六进制(hexfloat): 0x1.47ae14p-7
0.01 的默认格式(default): 0.01
3.14159
3.1415926000
3.1400000000
```

将 0x1P-1022 解析为十六进制的结果是：

```
2.22507e-308
```

再如：

```
#include <iostream>
#include <iomanip>
int main(){
    std::cout << "Left fill:\n" << std::left << std::setfill('*')
        << std::setw(12) << -1.23 << '\n'
        << std::setw(12) << std::hex << std::showbase << 42 << '\n'
        << std::setw(12) << std::put_money(123, true) << "\n\n";
    std::cout << std::setfill('#');
    std::cout << "Internal fill:\n" << std::internal
        << std::setw(12) << -1.23 << '\n'
        << std::setw(12) << 42 << '\n'
        << std::setw(12) << std::put_money(123, true) << "\n\n";

    std::cout << "Right fill:\n" << std::right
        << std::setw(12) << -1.23 << '\n'
        << std::setw(12) << 42 << '\n'
        << std::setw(12) << std::put_money(123, true) << '\n';
}
```

执行程序，输出结果：

```
Left fill:
-1.23*******
0x2a********
123*********

Internal fill:
-#######1.23
0x########2a
#########123

Right fill:
#######-1.23
```

```
########0x2a
#########123
```

输入运算符(>>)从输入流对象中读取指定类型的数据,如:

```
int a;  std::string s;
std::cin >> a >> s;
std::cout << a <<'\t'<< s;
```

执行上述代码:

```
3.14 hello
3       .14
```

这说明输入运算符先读取一个 int 类型的整数,然后读取一个字符串。默认情况下,输入运算符(>>)以空白字符(即空格、制表符、换行符、走纸、回车符)作为输入数据项的分隔符并忽略空白字符。

如果读取的是非法值或者遇到文件结束符,输入流对象将处于一个错误状态。当将输入流对象作为条件或循环语句的条件表达式时,会根据是否处于错误状态而隐式转换为 true 或 false。如果在上述代码后添加如下语句:

```
if (!std::cin)
    std::cout << "输入流处于错误状态!";
```

再执行代码:

```
hello 3
0       输入流处于错误状态!
```

这说明输入流对象处于错误状态,因为第一个输入的是字符串而不是一个整数。

13.1.3 非格式化输入输出

1. get()、put()和 getline()

和输入输出运算符一样,这些函数也都是以类模板的成员函数形式实现的。例如:

```
basic_ostream& put(char_type c);
```

put()函数模板将 char_type 字符类型的字符 c 输出到输出流对象,并返回输出流对象的自引用。下面以特定的 char 类型的类 istream 和 ostream 来说明这些成员函数。

(1) **get()**:从输入流中读取一个或多个字符。

这里有多个不同的版本。

int get():如成功读取一个字符,就返回这个字符的值。否则,返回 Traits :: eof()并设置流状态的标志位 failbit 和 eofbit。

istream& get(char& c):读入一个字符,存储在 c 中,并返回流对象的自引用。

istream& get(char * s,streamsize n,char delim)：获取 n−1 个字符或直到分隔符，将这些字符存储在 s 指向的字符数组中，并在最后添加结束字符'\0'。该函数将分隔符保留在输入流中。分隔符默认是换行符'\n'，但也可以通过 delim 指定。

istream& get(streambuf& sb,char delim)：从流中提取字符并将它们插入由流缓冲区对象 sb 控制的输出序列中，一旦插入失败或在输入序列中遇到分隔符 delim 时立即停止（分隔符默认是换行符'\n'），分隔符仍然留在输入流中。例如：

```
#include <iostream>
using namespace std;
int main(){
    char c;
    cout << "请输入:" << endl;
    while (cin.get(c))
        cout << c;
    return 0;
}
```

执行程序，当输入一行后，这一行的字符又被依次输出。直到用户按下 Ctrl+Z（Windows 平台）或 Ctrl+D(UNIX/Mac 平台）使输入流处于结束状态。

```
请输入:
B站和微博用户名: hw-dong
B站和微博用户名: hw-dong
博客网址: https://a.hwdong.com
博客网址: https://a.hwdong.com
```

下面代码从输入流对象 istringstream 中用 is.get(str,10,' ')每次最多读取 10 个字符，用空格字符作为分隔符以便读取空格字符分隔的单词。因为分隔字符仍然在输入流对象中，所以需要单独用 get()读取这个分隔符，才能继续读取后面的单词。

```
#include <iostream>
#include <sstream>
int main() {
    std::istringstream is("My name is hwdong");
    char str[10];
    while (is.get(str, 10, ' ')) {    //分隔符是空格字符
        is.get();                     //跳过分隔符
        std::cout << str << '\n';
    }
}
```

执行程序，输出结果：

```
My
name
is
hwdong
```

下面的程序先用 get()从 std::cin 读入文本到 256 字符数组 str 中，然后根据 str 表示的字符串文件名创建一个输入文件流对象 is，然后不断用 get()读取 is 中的单个字符直到遇到文件结束符。

```
#include <iostream>
#include <fstream>
int main() {
    char str[256];
    std::cout << "输入存在的文件名：";
    std::cin.get(str, 256);

    std::ifstream is(str);          //创建输入文件流对象 is
    char c;
    while (is.get(c))
        std::cout << c;

    is.close();                     //关闭打开的文件
}
```

执行程序，输出结果：

```
输入存在的文件名：test.txt
B站和微博用户名：hw-dong
博客网址：https://a.hwdong.com
哈罗 hello$ %346dsfsd@#
```

（2）**put**()：向输出流输出一个字符。

ostream& **put**(char c)：输出一个字符。

例如，下面的程序从键盘输入文本，直到遇到特殊字符，将输入的每个字符用 put()输出到一个文本文件 test.txt(其内容如图 13-5 所示)中：

```
#include <iostream>
#include <fstream>
int main() {
    std::ofstream outfile("test.txt");
    char ch;
    std::cout << "输入一些文本,直到遇到特殊字符#\n";
    do {
        ch = std::cin.get();
        outfile.put(ch);
    } while (ch != '#');
    return 0;
}
```



图 13-5 test.txt 文件的内容

执行程序,输出结果:

```
输入一些文本,直到遇到特殊字符#
B站和微博用户名: hw-dong
博客网址: https://a.hwdong.com
哈罗 hello$ %346dsfsd@#dfgsdg^
```

(3)**getline()**。

```
istream& getline(char* s,streamsize n);
istream& getline(char* s,streamsize n,char delim);
```

和 get()类似,最多读取 n−1 个字符到 s 指向的数组中直到遇到换行符或指定的分隔符,并追加一个结束字符'\0',但会抛弃流中的分隔符(默认是'\n',也可以指定分隔符)。

下面代码的 is.getline(str,8,'|')从 istringstream 对象中最多读取 8 个字符到 char 数组 str 中,遇到分隔符'|'结束。

```
#include <iostream>
#include <sstream>
int main() {
    std::istringstream is("hwdong|hw-dong|hw.dong");
    char str[10];
    while (is.getline(str, 8, '|'))
        std::cout << str << '\n';
}
```

执行程序,输出结果:

```
hwdong
hw-dong
hw.dong
```

下面代码读取文件 test.txt 中的内容并显示在控制台窗口中:

```
#include <iostream>
#include <fstream>
int main() {
```

```
    std::ifstream is("test.txt");
    char buf[1024];
    while (is.getline(buf, 1024))
        std::cout << buf << '\n';
}
```

执行程序,输出结果:

```
B站和微博用户名: hw-dong
博客网址: https://a.hwdong.com
哈罗 hello$ %346dsfsd@#
```

2. read()、write()和 gcount()

istream& **read**(char * buf,streamsize n):从输入流 istream 对象读取 n 个字符(字节)并保留在 char 数组 buf 中。与 get()和 getline()不同,它不会在读取内容最后追加结束字符'\0'。它主要用于读取二进制输入数据。

streamsize **gcount**() const:返回上次非格式化输入操作(如 get()、getline()、read()、ignore())读取的字符个数。

ostream& **write**(const char * buf,streamsize n):将 char 数组中的 n 个字节写(输出)到输出流对象中。

尽管这里是以 istream 和 ostream 为例说明的,这些方法也都适用于任何输入输出流,如文件输入输出流、字符串输入输出流。

例如,下面的程序每次 10 个字节块地读取文件 text.txt 的内容,并将读取的字符输出到 test2.txt 中。然后关闭文件输出流 os,再从 test2.txt 中最多读取 4096 个字符并输出到控制台窗口:

```
#include <iostream>
#include <fstream>
int main() {
      //binary 表示以二进制方式打开文件
    std::ifstream is("test.txt", std::ifstream::binary);
    std::ofstream os("test2.txt", std::ofstream::binary);
    char buf[10];
    while (is.read(buf, 10)) {
        os.write(buf, is.gcount());
    }
    os.write(buf, is.gcount());
    os.close();                    //关闭文件

    char buf2[4096];
    std::ifstream is2("test2.txt", std::ifstream::binary);
    is2.read(buf2, 4096);
    int num = is2.gcount();
    buf2[num] = '\0';
    std::cout << buf2;
}
```

执行程序，输出结果：

```
B站和微博用户名：hw-dong
博客网址：https://a.hwdong.com
哈罗 hello$ %346dsfsd@#
```

和 put()、get()、getline()会忽略分隔符不同，read()和 write()不会遗漏任何一个字符（字节）。

3. peek()和 putback()

它们是和读写操作相关输入流的 2 个成员函数。

charpeek()：返回输入流的下一个字符但不读取它。

istream& putback(char c)：将字符 c 插入回输入缓冲区。

例如，下面的程序先用 peek()看一下输入的第一个字符是否是 0～9 的数字，而采用不同的动作：

```
#include <iostream>
#include <string>
int main() {
    char c = std::cin.peek();
    if (c >= '0'&&c <= '9') {        //如果 cin 中的第一个字符是数字，则读入一个整数
        int num; std::cin >> num;
        std::cout << "你输入了一个整数：" << num << std::endl;
    }
    else {                           //否则，读入一行字符串
        std::string str;  getline(std::cin, str);
        std::cout << "你输入了一个字符串：" << str << std::endl;
    }
}
```

执行程序，输入一个字符串：

```
hello world
你输入了一个字符串：hello world
```

如果执行该程序，输入的是一个整数，如 4536：

```
4536
你输入了一个整数：4536
```

假如是用 get()读取一个字符，输入流缓冲区中就没有这个字符了，但可以用 putback()将这个字符再放回输入流缓冲区。上面的程序也可以这样写：

```
#include <iostream>
#include <string>
int main() {
    char c = std::cin.get();
```

```
        std::cin.putback(c);            //将字符 c 再放回去
        if (c >= '0'&&c <= '9') {
            int num; std::cin >> num;
            std::cout << "你输入了一个整数: " << num << std::endl;
        }
        else {
            std::string str;  getline(std::cin, str);
            std::cout << "你输入了一个字符串: " << str << std::endl;
        }
    }
```

13.1.4 文件位置

不同类型的输入输出流的辅助函数的数目是不一样的。例如，fstream 有成员函数 open()和 close()，用于打开和关闭一个文件，而 iostream（包括 istream 和 ostream）没有 open()和 close()函数。详细信息请参考官方文档。

istream 和 ostream 分别有成员函数 seekg()和 seekp()，用于重新定位文件位置，即分别用于设置输入流的读位置和输出流对象的写位置。例如：

```
basic_istream& seekg( pos_type pos );
basic_istream& seekg( off_type off, std::ios_base::seekdir dir);
basic_ostream& seekp( pos_type pos );
basic_ostream& seekp( off_type off, std::ios_base::seekdir dir );
```

其中，pos 表示相对于开头的绝对位置。off 是一个长整数，表示偏移量。dir 表示偏移量相对的位置，它有 3 个值：ios::beg(默认值)表示流的开头位置，ios::cur 表示流中的当前位置，ios::end 表示流的末尾位置。

这些函数都返回流对象自身的引用。

istream 和 ostream 分别有成员函数 tellg()和 tellp()返回当前文件指针的位置，即相对于文件开头偏移的字符(字节)数。例如：

```
pos_type tellg();
pos_type tellp();
```

下面的代码演示了 seekg()的用法：

```
#include <iostream>                 //std::cout
#include <fstream>                  //std::ifstream
int main() {
    std::ifstream is("test.txt", std::ifstream::binary);
    if (!is) return -1;
    //确定文件的长度
    is.seekg(0, is.end);            //定位到文件尾
    int length = is.tellg();        //查询长度
    is.seekg(0, is.beg);            //重新定位到文件开头
```

```
    //分配内存
    char * buffer = new char[length];
    //读取数据块
    is.read(buffer, length);
    is.close();
    //用 write()输出到 cout
    std::cout.write(buffer, length);
    delete[] buffer;
}
```

13.1.5　流状态

超类 ios_base 维护一个 iostate 类型的数据成员来描述流的状态，可以通过流的成员函数 rdstate()读取或 setstate()修改这个状态值。iostate 类型值是下列 const 成员(通过逻辑或运算 or 或|)的组合。

eofbit：当输入操作到达文件结尾时设置。

failbit：最后一个输入操作无法读取预期的字符或输出操作无法写入预期的字符。

badbit：由于 I/O 操作失败(例如文件读写错误)或流缓冲区导致的严重错误。

goodbit：没有上述错误，值为 0。

这些成员常量是 ios_base 中的公开静态成员，可以通过 ios_base :: failbit 等直接访问，也可以通过派生类(对象)访问，如 cin :: failbit 或 ios :: failbit。如：

```
std::ios_base::iostate s = cin.rdstate();
    cout << std::ios_base::badbit << '\t';
    cout << std::ios_base::failbit << '\t';
    cout << std::ios_base::eofbit << '\t';
    cout << std::ios_base::goodbit << std::endl;
    cout << s << std::endl;
```

执行程序，输出结果：

```
4    2    1    0
0
```

iostate 类型的状态值是 0 表示没有任何错误，即没任何错误时状态值的初始值就是 goodbit 的值。如果出现了 badbit 相关错误，状态值就是和 badbit 执行 or 运算的结果。如果出现了 fail 相关错误，状态值就是和 failbit 执行 or 运算的结果。可以通过下列程序模拟错误出现的过程：

```
std::ios_base::iostate s = cin.rdstate();
    cout << s << "\t";
    cin.setstate(std::ios_base::badbit);
    s = cin.rdstate();
    cout << s << "\t";
```

```
    cin.setstate(std::ios_base::failbit);
    s = cin.rdstate();
    cout << s << std::endl;
```

执行程序，输出结果：

```
0  4  6
```

使用如下 ios 类的公共成员函数可以方便地检测相应状态标志。

good()：如果设置了 goodbit，则返回 true(即没有错误)。

eof()：如果设置了 eofbit，则返回 true。

fail()：如果设置了 failbit 或 badbit，则返回 true。

bad()：如果设置了 badbit，则返回 true。

clear()：清除 eofbit、failbit 和 badbit。

例如，下列程序试图打开一个不存在的文件，其标志位 failbit 会被设置。

```
#include <iostream>
#include <fstream>
int main() {
    std::ifstream is("tesy.txt");
    if ((is.rdstate() & std::ifstream::failbit) != 0)
        std::cerr << "打开 'test.txt'出错\n";
    std::cout << is.good() << "\t" << is.eof() << "\t"
            << is.fail() << "\t" << is.bad() << "\n";
}
```

执行程序，输出结果：

```
打开 'test.txt'出错
0    0    1    0
```

当一个流处于错误状态时就无法进行输入输出，流对象根据其状态可自动转换为 bool 类型的值，即如果流处于错误状态，流对象可转换为 false，反之，则转换为 true。因此，可将流对象作为一个条件表达式。如：

```
std::string word;
while(std::cin >> word)
//...
```

再如：

```
std::isftream ifile("test.txt")
    if(!ifile) return false;
```

另外，当流处于错误状态时，可用 clear()将这个状态标志设置为 0，即没有任何错误的状态，从而可以继续执行输入或输出操作。

13.1.6 管理输出缓冲区

每个输出流都管理一个输出缓冲区，这是一块计算机内存，输出的信息通常先放到这个输出缓冲区中而不是直接输出到输出设备，从而可以提高程序的效率。试想如果 put(c) 每次调用都直接对外部物理设备执行速度慢的写操作，效率会有多低？通过设置输出缓冲区，允许操作系统可以一次性将多个输出组合成单个的实际设备的输出操作，大大提高程序的性能。

导致缓冲区刷新（即数据真正写到物理设备上）的原因有很多，具体如下。

（1）程序正常结束，作为 main() 函数 return 语句结束的一部分，会执行缓冲区刷新。

（2）缓冲区已满（包含正常情况和异常情况），需要刷新缓冲区，数据才能继续写入缓冲区。

（3）使用操纵符显式地刷新输出缓冲区，如：endl、ends、flush。

（4）使用 unitbuf 操纵符设置流的内部状态，来清空缓冲区。默认情况下，对 cerr 是设置 unitbuf 的，即对 cerr 的输出会立即刷新。

（5）输出流可能被关联到另一个流，这种情况下，对另一个流的读写会立即导致被关联输出流的刷新。默认情况下，cin 和 cerr 都关联到 cout，因此，读 cin 和写 cerr 都会立即刷新 cout。

例如，下面的前 3 条语句都会导致刷新输出缓冲区：

```
std::cout << "hi" << std::endl;      //输出 hi 和换行符,然后刷新缓冲区
std::cout << "hi" << std::flush;     //输出 hi,然后刷新缓冲区
std::cout << "hi" << std::ends;      //输出 hi 和空字符,然后刷新缓冲区
std::cout <<"world\n";
```

代码的输出是：

```
hi
hihi world
```

13.1.7 文件输入输出

1. 文件输入输出流

前面看到，文件输入输出流和标准输入输出流的使用是类似的。在头文件< fstream >中，类 ofstream 是 ostream 的子类，类 ifstream 是 istream 的子类，类 fstream 是 iostream 的子类，用于双向 I/O。要使用这些类，需在程序中包含< fstream >头文件。

在定义文件输入输出流对象时，如果给构造函数传递一个文件名（文件路径），文件流对象将直接打开这个文件。如果没有传递文件名，将创建不关联任何文件的流对象，之后通过成员函数 open(filename) 关联并打开文件 filename。

可以用文件流的 close() 成员函数关闭（断开）文件流对象关联的文件。当一个文件流对象被销毁时，其关键的文件也会自动被关闭。例如：

```
#include <iostream>                 //std::cout
#include <fstream>                  //std::ifstream
int main() {
    std::ofstream os;
    os.open("test3.txt");           //输出文件流对象 os 关联并打开文件"test3.txt"
    os << "hello world!  in test3\n";
    os.close();                     //关闭(断开)文件流对象关联的文件
    os.open("test4.txt");           //再关联并打开另外一个文件"test4.txt"
    os << "hello world!  in test4\n";
}
```

上述程序创建一个文件输出流对象 os,接着用 open()关联并打开一个文件,如果不存在这个文件,则会创建一个文件,如果已经存在,默认会清空文件中的内容。用输出运算符<<输出一串字符后就用 close()关闭了该文件。接着再用 open()关联并打开另外一个文件,最后程序结束销毁 os 时,这个文件会自动被关闭。

2. 文件打开模式

每个流都有一个文件模式(file mode),指出如何使用文件。

文件模式在 ios_base 超类中定义为静态公共成员。可以从 ios_base 或其子类访问它们,通常使用子类 ios。可用的文件模式标志如下。

ios :: in:以输入模式打开文件。

ios :: out:以输出模式打开文件。

ios :: app:输出附加在文件的末尾。

ios :: trunc:截断文件并丢弃旧内容。

ios :: binary:用于二进制(原始字节)I/O 操作,而不是基于字符的操作。

ios :: ate:将文件指针定位到文件末尾位置。

这些标志并不是互相排斥的,可以通过位或(|)运算符设置多个标志,例如,ios :: out | ios :: app 表示在文件末尾追加写的方式打开文件。对于输出,默认值为 ios :: out | ios::trunc,即以写模式打开文件且清空文件内容。对于输入,默认值为 ios :: in,即以读的方式打开文件。

要指定文件模式,需要设置 open()函数的第二个形参即文件模式形参的值或者在创建流对象传递文件名时设置这个文件模式。即:

```
std::ifstream iF;
iF.open(filename, mode);            //以 mode 模式打开文件
//......
iF.close();
```

或

```
std::ifstream iF(filename, mode);   //以 mode 模式打开文件
```

例如：

```
#include <iostream>                 //std::cout
#include <fstream>                  //std::ifstream
#include <string>
int main() {
    std::ifstream iF;
    iF.open("test3.txt", std::ifstream::in);
                                    //或 std::ios_base::in 以输入模式打开文件
    std::string str;
    while (iF >> str)std::cout << str ;

    std::cout << std::endl;
    std::ifstream iF2("test4.txt", std::ios_base::binary);
                                    //再以二进制模式打开另外一个文件
    char buf[10];
    while (iF2.read(buf, 10)) {
        std::cout.write(buf, iF2.gcount());
    }
    std::cout.write(buf, iF2.gcount());
}
```

执行程序，输出结果：

```
helloworld!intest3
hello world! in test4
```

13.1.8 字符串流

C++通过头文件<sstream>中的字符串流类，使得可以用相同的流公共接口来支持程序和字符串流对象(缓冲区)之间的输入输出。即以流的方式将数据输出到一个字符串流对象或者从一个字符串流对象读取数据。

<sstream>包含的字符串流类有 istringstream(istream 的子类)、ostringstream(ostream 的子类)和双向 stringstream(iostream 的子类)。

```
typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_stringstream<char> stringstream;
```

字符串输入流可用于解析或验证输入数据，字符串输出流可用于格式化输出。

1. istringstream

定义 istringstream 流对象可以有一个初始的 string 值，也可以没有；可以设置模式(mode)。

```
explicit istringstream(ios::openmode mode = ios::in);  //默认空的 string
explicit istringstream(const string & buf, ios::openmode mode = ios::in);
                                    //有一个初始化的 string 值即 buf 的值
```

例如：

```
#include <iostream>                    //std::cout
#include <sstream>                     //std::istringstream
#include <string>                      //std::string
int main() {
    std::string stringvalues = "123 3.14 hello";
    std::istringstream iss(stringvalues);  //创建一个字符串输入流对象 iss
    int i; double f; std::string s;
    iss >> i >> f >> s;                //从输入流对象 iss 中读取数据
    std::cout << i << '\t' << f << '\t' << s << '\n';
}
```

可以和其他输入流(如 std::cin)或文件输入流一样,从输入字符串流对象(如上述代码中的 iss)用流操作的接口如>>、getline()等读取数据,唯一不同的是数据来自于内存而不是键盘或外部文件。

执行程序,输出结果:

```
123    3.14    hello
```

2. ostringstream

同样,输出字符串流可以将一个 string 当作通常的输出流对象一样使用。即向这个输出字符串流对象输出一定格式的数据。

其构造函数如下:

```
explicit ostringstream(ios::openmode mode = ios::out);     //默认空的 string
explicit ostringstream(const string & buf,
            ios::openmode mode = ios::out);                //有一个初始化的 string
```

获取或设置其中的 string 值的 2 个成员函数 str():

```
string str() const;                                        //得到流对象存储的 string 值
void str(const string & str);                              //设置流对象的 string 值
```

例如:

```
#include <iostream>                                        //std::cout
#include <sstream>                                         //std::ostringstream
#include <string>                                          //std::string
int   main() {
    std::ostringstream sout;                               //构造字符串输出流对象
    sout << "zhang" << "," << 80.5;                        //写数据到字符串输出流对象
    std::cout << sout.str() << std::endl;                  //获取内容
    sout.str("hello world");                               //设置内容
    std::cout << sout.str() << std::endl;                  //获取内容
}
```

执行程序，输出结果：

```
zhang,80.5
hello world
```

13.2 容器

13.2.1 标准容器

容器库是一组模板和算法，它们实现了通用的数据结构。容器是存储一组数据元素的对象。每个容器都管理其数据元素的存储空间，并通过迭代器和成员函数提供对每个元素的访问。

标准容器(standard containers)实现了如下常用的数据结构。

- 动态数组(dynamic arrays)：如 array、vector。
- 队列(queues)：如 queue。
- 堆栈(stacks)：如 stack。
- 链表(linked lists)：如 forward_list、list。
- 树(trees)：没有真正的树的实现，但 map、set 的内部采用了树的结构。
- 关联集合(associative sets)：如 map、set 等。
- 哈希表(散列表)：如 unordered_map、unordered set 等。

C++容器库的类可分为如下 4 种。

- 序列容器(sequence containers)。
- 容器适配器(container adapters)。
- 关联容器(associative containers)。
- 无序关联容器(unordered associative containers)。

1. 序列容器

- array：静态连续的数组。数组一旦初始化后，就不能改变大小。
- vector：动态连续的数组。数组的空间容量和大小可动态改变。
- forward_list：前向的单链表。只能从开头到结尾方向前进。
- list：双向链表。可正向反向前进。
- deque：双向队列。可在队列的前端和后端插入或删除数据元素。

std::string 虽然不属于 STL 的序列容器，但也满足序列容器需求。所谓序列容器需求是指该容器必须实现下列方法：back()、push_back()和 pop_back()。其中，back()表示查询最后一个元素，push_back()表示将一个元素加入序列容器最后，pop_back()表示删除最后一个元素。

2. 容器适配器

容器适配器是一种特殊类型的容器类。它们本身不是完整的容器类，而是其他容器类型(例如 vector、deque 或 list)的包装器。这些容器适配器封装底层容器类型并相应地限制

了用户接口。

例如,std::stack 是一个施加了"后进先出 LIFO"特性的数据结构。其声明如下:

```
template < class T, class Container = std::deque < T >> class stack;
```

容器 std::stack 是序列容器,如 std::deque < T >容器的包裹器。std::deque 是默认的底层容器,当然也可以用其他序列容器如 std::vector 和 std::list 代替 std::deque。

标准容器适配器有如下几种。

- stack:"后进先出 LIFO"特性的数据结构。
- queue:"先进先出 FIFO"特性的数据结构,普通的队列。
- priority_queue:优先队列。因为元素是按照优先值大小排列的,因此,可以不需要逐个比较,(在恒定时间内)直接定位到最佳元素(默认情况下)。

3. 关联容器

关联容器按"键"(关键字)存储数据元素,可以通过"键"快速查找一个数据元素(时间复杂度 $O(\log n)$)。关联容器类型可以分为 2 类:键唯一的关联容器,以及同一键多值的关联容器。

1) 键唯一的关联容器

(1) set:键的集合,按键排序。

(2) map:"键-值"对的集合,按键排序。

2) 同一键多值的关联容器

(1) multiset:键的集合,按键排序。

(2) multimap:"键-值"对的集合,按键排序。

这些关联容器通常用红黑树实现。

每个关联容器都可以在声明时指定一个比较函数。如 std :: set 的定义:

```
template < class Key,
    class Compare = std::less < Key >,
    class Allocator = std::allocator < Key >> class set;.
```

默认比较函数是 std :: less。比较函数用于对键进行排序。可以指定不同的比较函数。

4. 无序关联容器

无序关联容器提供可以哈希访问的未排序数据结构。在最坏的情况下,访问时间是 $O(n)$,但是对于大多数操作来说,访问时间比线性时间少得多,可以认为是常数时间 $O(1)$。

对于所有无序关联容器类型,使用哈希键(hashed key)来访问数据。与关联容器类似,它分为键唯一的无序关联容器和同一键多值的无序关联容器。

1) 键唯一的无序关联容器

(1) unordered_set:键的集合,根据键的哈希存储。

(2) unordered_map:"键-值"对的集合,根据键的哈希存储。

2) 同一键多值的无序关联容器

(1) unordered_multiset:键的集合,根据键的哈希存储。

(2) unordered_multimap:“键-值”对的集合,根据键的哈希存储。

与其他容器类型一样,无序关联容器类型也是模板,例如 std :: unordered_set。

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>> class unordered_set;
```

程序员可以传递自定义的哈希函数、键比较函数和(内存)分配器。

下面通过一些具体类型的使用例子来理解这些容器。

13.2.2 序列容器

序列在数据结构中也称为线性表,即是一组数据元素的线性排列:(a_1,a_2,…,a_n)。C++的序列容器是序列(线性表)的具体实现。

array 和 vector 都是采用动态分配的一块连续存储空间来存储数据元素,即逻辑上相邻的数据元素在物理地址上也是相邻的(如 a_2 和 a_3 地址上是紧靠的),但 array 的空间不会动态增长,即 array 初始化确定大小(数据元素个数)后就再不能改变空间大小,而 vector 存储数据元素的空间可以动态增长,可以向 vector 添加任意多个数据元素,只要计算机内存足够大。因为采用连续存储方法,因此,它们具有随机存储的优点,即可以通过下标运算符[]或 at()函数在常数时间 $O(1)$存取某个数据元素,速度非常快。对于可以插入/删除数据元素的 vector,在中间某个位置进行插入 insert()和删除 erase ()操作,需要移动后面的所有元素,速度较慢($O(n)$数量级),但在最后插入 push_back()、删除 pop_back()数据元素的速度和随机存取元素一样快,也是常数时间 $O(1)$。std::array 是对 C++内存数组的轻量级包装,隐藏了底层的指针并提供了一些有用的成员函数。对于不需要动态改变大小的线性表(数组),比 vector 有更高的效率。

forward_list 和 list 都是采用链表结构存储序列中的数据元素,每个元素对应一个单独的内存块,不同元素之间的内存块在计算机内存中的位置是任意的、散落的,即逻辑上相邻的数据元素在物理地址上通常并不相邻。每个元素的内存块里通过指针指向逻辑上相邻元素的内存块,即用指针将这些内存块串在一起,表示逻辑上的相邻关系。forward_list 的数据元素的内存块只有一个**前向指针**,指向逻辑上下一个(直接后继)数据元素的内存块地址,而 list 的每个数据元素的内存块有 2 个指针,分别指向逻辑上直接前驱和直接后继数据元素的地址。list 和 forward_list 的主要优点是在中间插入或删除元素,不需要移动其他数据元素,但需要先定位到插入或删除的位置,速度也较快。另外在首尾插入或删除数据元素的操作 push_front()、push_back()、pop_front()、pop_back()的速度很快($O(1)$),但 forward_list 的 pop_back()和中间位置插入的速度类似。

deque 的存储结构混用了链表和连续存储 2 种方式,使得可以以常数时间在序列的首尾两端插入和删除数据元素。

下面通过一些具体例子来理解这些容器。

1. std::vector

常用操作是在尾部的插入 push_back()和删除 pop_back()、通过下标运算符[]或 at()方法访问某个下标的数据元素(下标从 0 开始,对于一个 vector 对象 vec,其最大下标是 vec.size()－1)。at()和[]的区别是 at()方法会检查下标是否在合法的范围内。

下面的代码说明这些操作的使用以及创建 vector 实例化类对象的不同方式。

```
#include <iostream>
#include <vector>                          //包含 vector 类模板的头文件
int main() {
    std::vector<int> vec;                  //空的 vector 对象
    std::vector<int> vec2(3);              //3 个元素,每个元素的初始值为 0
    std::vector<int> vec3{ 3 };            //初始化列表,只有一个元素 3
    std::vector<int> vec4{ 1,3,5,7,9 };    //初始化列表,包含 5 个元素
    std::vector<int> vec5(vec4);           //拷贝构造函数

    vec = vec5;                            //赋值运算符

    vec.pop_back();                        //删除最后的元素,即 9
    vec.push_back(2);                      //push_back()将 2 加到最后面
    vec.push_back(4);
    vec[0] = 100;                      //通过下标运算符访问某个元素
    vec.at(1) = 200;                   //类似于下标运算符[],at()方法通过下标访问某个元素
                                       //at()会检查下标是否超出范围,而[]不检查
    for (auto i{ 0 }; i < vec.size(); i++) //size()函数返回 vec 的大小(元素个数)
        std::cout << vec[i] << '\t';
    std::cout << std::endl;
}
```

执行程序,输出结果:

```
100     200     5       7       2       4
```

除了用下标外,也可以通过 range for 遍历 vector:

```
void print(const std::vector<int> &vec) {
    for (auto e : vec)
        std::cout << e << '\t';
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec(3);
    for (auto& e : vec)                //e 是引用 vec 的元素,才能修改它
    e = 1;
    vec.push_back(10);
    print(vec);
}
```

执行程序,输出结果:

```
1          1          1          10
```

除了通过下标访问数据元素外，vector 和所有容器一样，可以通过类似于指针的迭代器访问数据元素，begin()和 end()方法分别返回指向第一个元素位置和最后一个元素后一个位置的迭代器，可以通过解引用运算符 * 得到一个迭代器指向的数据元素（引用）。可以在 vector 的迭代器指示位置插入或删除数据元素。

迭代器类似于指针，可以和整数相加减对迭代器进行偏移。

```
int main() {
    std::vector<int> vec{ 1,2,3,4,5,6 };
    std::cout << *(vec.begin()) << '\t';     //vec.begin()返回指向 vec 开头的迭代器
                                             //迭代器类似于指针,可以用 * 得到它指向的元素
                                             //输出结果: 1
    std::cout << *(vec.begin() + 2) << '\t'; //对迭代器加上一个整数,使得迭代器偏移
    std::cout << *(vec.end() - 2) << '\t';   //对迭代器减去一个整数,使得迭代器偏移
    auto p = vec.begin();
    p++;                                     //迭代器可以自增或自减,指向第 2 个元素
    std::cout << *p << '\n';
    print(vec);
    vec.insert(vec.begin() + 1,100);         //在迭代器指示位置插入一个新元素 100
    vec.erase(vec.end() - 1);                //删除最后一个元素
    print(vec);
}
```

执行程序，输出结果：

```
1        3        5        2
1        2        3        4        5        6
1        100      2        3        4        5
```

和 push_back() 一样，emplace_back()可在最后添加一个元素，区别是 emplace_back()可以在相应数据元素的内存块里直接构造一个数据元素，而不需要将一个其他地方创建好的数据元素复制到这个数据元素的内存块，从而提高了效率，特别适用于占用内存空间比较大的数据类型。emplace()和 insert()类似，可以在迭代器位置插入数据元素，且和 emplace_back()也类似，可以在对应位置直接创建数据元素。

```
int main() {
        std::vector<int> vec;
        vec.emplace_back(1);
        vec.emplace(vec.begin() + 1, 2);    //在迭代器位置 vec.begin() + 1 的内存里直接创建
                                            //一个数据元素
        print(vec);
}
```

执行程序，输出结果：

```
1        2
```

另外，还可以通过 data()返回存储实际数据元素的内存块的地址(指针)，然后通过指针访问 vector 的数据元素。

```
int main() {
    std::vector<int> vec{1,2,3,4};
    int *p = vec.data();                //data()返回存储实际数据元素的内存块的地址(指针)
    while (p != vec.data() + vec.size()) {      //通过指针遍历
        *p *= 2;
        p++;
    }
    print(vec);                         //输出结果: 2        4        6        8
    vec.clear();                        //清空
    vec.reserve(5);                     //容量变为 5
    std::cout << "大小: " << vec.size() << '\t'
        << "容量: " << vec.capacity() << '\n';   //输出: 大小: 0 容量: 5
}
```

执行程序，输出结果：

```
2        4        6        8
大小: 0        容量: 5
```

vector 类模板的各种方法可以在网上搜索到，也可以在集成开发环境如 Visual Studio 的代码编辑器中输入变量名和小数点如"vec."，就会显示一个方法列表提示，如图 13-6 所示。

C++的不同容器的同样操作的方法名都是一样的，熟悉了 vector 的方法，对于其他容器类型的对象，就可以尝试用同名方法执行同样的操作。



图 13-6 Visual Studio 2017 的智能提示

2. std::array

std::array 是类似于 vector 的占用连续内存的数组，但其大小在定义时就确定了，以后不能动态改变大小，这点类似于 C++自带的数组，但 std::array 是一个容器类模板，提供了许多对容器对象进行操作的方法。因为不能动态改变大小，自然就没有插入或删除数据元素的方法，如 push_back()、insert()、erase()等。

类模板 array 有 2 个模板参数：一个类型模板参数说明数据元素的类型；另一个非类型模板参数说明数组的大小。因此，array 的实例化必须传递 2 个模板实参：

```
array<typename T, int N>
```

不同模板参数实例化的 2 个不同 array 类(如 std::array < int,5 >和 std::array < int,4 >)

是完全不同的 2 个类型，它们的对象是不能相互赋值的。

```
#include <array>
int main() {
    std::array<int, 5> arr;              //必须指定 array 的大小
    std::array<int, 4> arr2{1,3,7};
    arr[4] = 3;
    arr2.at(3) = 13;
    for(auto e:arr)
        std::cout << e << '\t';
    std::cout << std::endl;
    std::cout << arr2.back() << std::endl;
    //  arr = arr2;                      //错：不能相互赋值.arr 和 arr2 属于完全不同的类型
                                         //std::array<int, 5>和 std::array<int, 4>
}
```

执行程序，输出结果：

```
-858993460      -858993460      -858993460      -858993460      3
13
```

3. std::deque

std::deque 是一个双向队列(double-ended queue)，可以在两端插入(push_front()、push_back())或删除数据元素(pop_front()、pop_back())，并且可以通过下标(下标运算符[]或 at()方法)去访问数据元素。

```
#include <deque>                         //双向队列(double-ended queue)
int main() {
    std::deque<int> deq{ 1,3 };
    deq.push_back(5);
    deq.push_front(-3);
    deq.push_front(-5);
    deq[1] = 10;
    deq[4] = 40;
    for (auto e : deq)
        std::cout << e << '\t';
    std::cout << std::endl;

    deq.pop_front();
    for (auto e : deq)
        std::cout << e << '\t';
    std::cout << std::endl;
}
```

执行程序，输出结果：

```
-5      10      1       3       40
10      1       3       40
```

4. std::list

因为 list 是链表，数据元素是单独存放的，所以所有数据元素不是存储在一块连续存储空间。因此，list 不同于数组，不能用下标运算符去访问其中的元素，只能用迭代器去访问它的元素。

下面是 std::list 的一个示例。

```
#include <list>                          //单向链表 list 类模板的头文件
int main() {
    std::list<int> l;                    //空的 list
    std::list<int> l2{2,3,5};            //列表初始化
    std::list<int> l3(1);
    //  l3[1] = 3;                       //错：不能用下标运算符

    l = l3;
    l.push_front(-3);
    l.push_back(3);
    l.insert(l.begin(), -5);             //begin()和 end()返回第一个元素和最后元素的
                                         //后一个位置的迭代器
                                         //迭代器类似于指针,可以用 * 得到它指向的元素
    for (auto it = l.begin(); it != l.end(); it++)
        std::cout << *it << '\t';
    std::cout << std::endl;              //输出: -5      -3      0       3

    for (auto it = l.begin(); it != l.end(); it++)
        *it *= 3;
    for (auto it = l.begin(); it != l.end(); it++)
        std::cout << *it << '\t';
    std::cout << std::endl;              //输出: -15     -9      0       9

    auto it = l.end();
    it--; it--;
    l.insert(it, 100);
    l.pop_back();
    for (auto it = l.begin(); it != l.end(); it++)
        std::cout << *it << '\t';
    std::cout << std::endl;              //输出: -15     -9      100     0
    return 0;
}
```

13.2.3 容器适配器

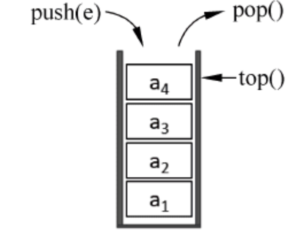
std :: stack <>、std :: queue <>和 std :: priority_queue <>被称为容器适配器，因为它们本身在技术上不是容器。它们是序列容器的包裹器，它们包裹的序列容器默认情况下是 std::vector <>或 std::deque <>，它们利用底层的容器来实现一组特定的、操作受限的成员函数。例如，std::stack 利用 deque <>作为底层容器存储数据元素，即 std::stack 有一个 std::deque <>类型的私有变量，但 std::stack 只提供在一端进行插入(push())、删除

(pop())、查询(top())的方法,使得外界无法对其底层的 std::deque<>变量进行其他操作(如中间、两端的插入或删除)。

stack(栈)类似于餐馆里的一叠碟子,如图 13-7 所示,新的碟子只能放在最上面,同样也只能从最上面拿走一个碟子。因此,它是一种具有先进后出(FILO)或者后进先出(LIFO)特性的数据结构。stack 的插入、删除或查询只能在一端进行,该端称为"栈顶"。例如 top()方法用来查询栈顶元素。



(a) 一叠盘子就是一个堆栈



(b) 入栈 push、出栈pop、取栈顶top

图 13-7 栈具有 FILO 或 LIFO 特性

类似于 std::stack<>,std::queue<>(队列)也是序列容器的适配器(包裹器)类模板,表示的是一种具有先进先出(FIFO)特性的数据结构。类似于日常生活中的各种排队,后来的人总是排到最后,队头的人总是先出。

std::priority_queue<>类似于 std::queue<>,但每个数据元素具有一个优先级,优先级高的总是排在队列的前面,如银行的 VIP 客户总是比普通客户优先。

这些对序列容器包裹的容器适配器通过对原有容器的操作做了限制,对于特定的应用场合,使用这些容器适配器更能保证数据的安全性、不容易出错。

std::priority_queue<>和 std::queue<>经常用于任务调度,如一个应用程序有各种事件(键盘、鼠标、网络等)消息,操作系统会将所有的消息放入应用程序的消息队列,应用程序通过查询这个消息队列依次对这些消息处理。再如一个操作系统中有很多程序(称为进程),操作系统会按照进程的优先程度不同将这些进程放入一个优先队列,然后按照优先次序将它们调度到 CPU 中去执行。

```
#include <iostream>
#include <stack>
#include <queue>
int main(){
    //stack 的使用
    std::stack<int> stack;
    for (int i{}; i<10; ++i)
        stack.push(i);
    std::cout << "不断从栈顶 pop()出栈元素,直到栈为空\n";
    while (!stack.empty())
    {
        std::cout << stack.top() << ' ';    //top()获取栈顶元素
        stack.pop();                        //pop()方法不返回任何值
    }
    std::cout << std::endl;
```

```
    //queue 的使用
    std::queue<int> queue;
    for (int i{}; i<10; ++i)
        queue.push(i);
    std::cout << "不断从队头 pop()出队元素,直到队列为空: \n";
    while (!queue.empty())
    {
        std::cout << queue.front() << ' ';   //front()获取队头元素
        queue.pop();                         //pop()方法不返回任何值
    }
    std::cout << std::endl;
}
```

上述程序通过 push()将一个数据元素入栈或入队,通过 pop()将栈或队列中元素出栈或出队。可以看到 stack 出栈的序列和入栈的序列正好相反,而 queue 出队和入队的序列是一样的。

执行程序,输出结果:

```
不断从栈顶 pop()出栈元素,直到栈为空
9 8 7 6 5 4 3 2 1 0
不断从队头 pop()出队元素,直到队列为空
0 1 2 3 4 5 6 7 8 9
```

13.2.4 关联容器

1. set、unordered_set、multiset 和 unordered_multiset

set(集合)是一个数据元素只能出现 1 次的容器(数据结构)。通过 insert()方法向其中添加一个元素,如果要添加的元素已经存在于 set 中,则不会添加到 set 中。通过 erase()方法将一个数据元素从 set 对象中删除(擦除)。

```
#include <iostream>
#include <set>

void print(const std::set<int>& S) {
    for (int e : S)
        std::cout << e << '\t';
    std::cout << std::endl;
}
int main() {
    std::set<int> s;
    s.insert(1);                    //插入
    s.insert(2);
    s.insert(3);
    s.insert(4);
    s.insert(1);
    s.erase(2);                     //删除
    print(s);
```

```
    s.clear();
    s.insert(2);
    print(s);
}
```

执行程序，输出结果：

```
1       3       4
2
```

可以用 count()或 find()方法检查一个元素是否在集合中，前者返回值 1 或 0，后者返回一个迭代器，需要将这个迭代器和 end()返回的迭代器比较，判断是否存在这个元素。在上述代码后添加如下代码：

```
if (s.find(2) != s.end())
        std::cout << "2 在集合 s 中\n";
    std::cout << s.count(2) << '\t' << s.count(1) << '\n';
```

执行程序，输出结果：

```
2 在集合 s 中
1       0
```

注意：set 没有 push_back()或 pop_back()这些序列容器或适配器的方法。

set 是一个有序关联容器，数据元素之间能比较大小，默认使用的 std::less < T >，即数据元素类型 T 能用<运算符比较大小。也可以在实例化 set 时提供定制的比较函数或谓词。set 内部用一个类似平衡二叉树的红黑树按照元素的大小有序地存储，因此，不管按照什么次序输入一组值，存储的次序都是一样的。

例如：

```
int main() {
    std::set < int > s;
    s.insert(3);                    //插入
    s.insert(1);
    s.insert(2);
    print(s);
    std::set < int > s2;
    s2.insert(2);                   //插入
    s2.insert(3);
    s2.insert(1);
    print(s2);
}
```

执行程序，输出结果：

```
1       2       3
1       2       3
```

如果用 std::greater<>作为比较谓词：

```
int main() {
    std::set<int> s;
    s.insert(3);                        //插入
    s.insert(1);
    s.insert(2);
    print(s);
    std::set<int,std::greater<>> s2;
    s2.insert(2);                       //插入
    s2.insert(3);
    s2.insert(1);
    for (int e : s2)
        std::cout << e << '\t';
    std::cout << std::endl;
}
```

执行程序，输出结果：

```
1       2       3
3       2       1
```

因为采用的是红黑树，因此，set 的插入、删除、查找的时间复杂度是 $O(\log n)$。

和 set 不同，unordered_set 是无序的关联容器，数据元素之间无须比较大小，其内部是采用 hash 表的数据结构存储所有数据元素的，即对于每个元素，通过 hash 函数算出其存储地址，然后根据这个地址进行存取操作，因此，存取速度非常快，理想情况下时间复杂度是 $O(1)$，即常数时间就能直接定位到数据元素的存储地址。有兴趣的读者可以测试一下采用 set 和 unordered_set 的存取数据元素的速度。

set 和 unordered_set 不能存储重复的数据元素，而 multiset 和 unordered_multiset 可以存储重复元素，即多个数据元素的键值可以是相同的。因此，count()函数的返回值可以大于 1。它们存取数据元素的速度也很快。

2．map

map（也称**关联数组**）是一种以“**键-值**”形式存储数据元素的容器，即根据**键**（**关键字**）来存储数据元素的值。它描述的是如字典、电话簿这类根据键查找值的数据结构。如根据一个人名查找电话号码。键必须是唯一的，不同键的值可以相同。如一个单位的人可以用同一个电话号码。

类似于 set<T>和 unordered_set<T>，有 2 种 map（关联数组）：std::map<Key，Value>和 std::unordered_map<Key，Value>。前者是有序的关联数组，后者是无序的关联数组。和其他容器不同，map 必须有 2 个模板参数：一个类型模板参数说明键（key）的数据类型；另一个类型模板参数说明值的数据类型。

```
#include <iostream>
#include <string>
int main(){
    std::map<std::string, unsigned long long> phone_book;
```

```
    phone_book["Li Ping"] = 13101966886;
    phone_book["Zhang Wei"] = 15301966686;
    phone_book["Wang qiang"] = 13101966886;
    phone_book["Pan Xiao"] = 12401966888;
    std::cout << "Li Ping 的电话号码是: " << phone_book["Li Ping"] << std::endl;
    for (const auto&[name, phone] : phone_book)
        std::cout << name << " 的号码: " << phone << std::endl;
}
```

该程序中定义了一个键类型 std::string,值类型是 unsigned long long 的 map 对象。可以通过下标运算符[]对某个键的值进行存取(读写)。

执行程序,输出结果:

```
Li Ping 的电话号码是: 13101966886
Li Ping 的号码: 13101966886
Pan Xiao 的号码: 12401966888
Wang qiang 的号码: 13101966886
Zhang Wei 的号码: 15301966686
```

同样,可以用 count()或 find()查找是否存在某个键对应的元素:

```
if (phone_book.find("Zhang Wei") != phone_book.end())
    std::cout << "Zhang Wei 的号码是: "<< phone_book ["Zhang Wei"]<<'\n';
std::cout << phone_book.count("Zhang Wei")<<'\t'
        << phone_book.count("ZhangWei")<< '\n';
```

执行程序,输出结果:

```
Zhang Wei 的号码是: 15301966686
1       0
```

13.3 迭代器

13.3.1 迭代器及其分类

某些容器类型(如 array、vector、deque)可以通过下标去访问其中的数据元素或者通过成员函数 data()返回的原始指针去访问数据元素。但下标或原始指针不适合大部分容器类型,为此,C++标准库的所有容器都通过所谓的**迭代器**(iterator)提供了统一的访问容器中数据元素的接口。即每个容器都定义了嵌套的**迭代器**类型用于访问这个容器类型的数据元素。

迭代器本身不是数据元素而是表示数据元素的存储位置。类似于指针,可以通过解引用运算符 * 访问迭代器指向的数据元素。

1. const、non-const 迭代器和逆向迭代器

对于一个数据类型 T,T * 和 const T * 分别表示的是指向 non-const 对象和 const 对象

的指针。同样,容器的迭代器也分为 2 种类型:指向 non-const 和 const 对象的迭代器。例如:

```
std::vector<int>::iterator it;
std::vector<int>::const_iterator cit;
```

std::vector<int>::iterator 和 std::vector<int>::const_iterator 分别是 std::vector<int>类内部的 non-const 和 const 迭代器类型。it 和 cit 分别是这两种类型的对象。it 是指向 non-const 对象的迭代器,可以通过 it 修改它指向的数据元素;而 cit 是指向 const 对象的迭代器,不能通过 cit 修改它指向的数据元素。

容器的 begin()和 cbegin()成员函数返回的是指向容器的第一个元素的迭代器,而 end()和 cend()成员函数返回的是指向最后一个元素的后一个位置的迭代器。注意,end()-1 或 cend()-1 指向的才是最后一个元素的迭代器。因此,可以用 begin()或 cbegin()分别初始化 it 或 cit,然后通过自增运算符不断让迭代器向最后一个元素方向移动。

还可以用逆向迭代器逆向访问数据元素。例如,std::vector<int>:: reverse_iterator 和 std::vector<int>::const_reverse_iterator 分别是逆向的 non-const 和 const 迭代器类型。对这 2 种迭代器类型对象执行自增运算符则会沿着序列容器的逆向前进。rbegin()和 crbegin()返回指向序列尾部元素的逆向迭代器,而 rend()和 crend()返回指向序列第一个元素的前一个位置的逆向迭代器,如图 13-8 所示。

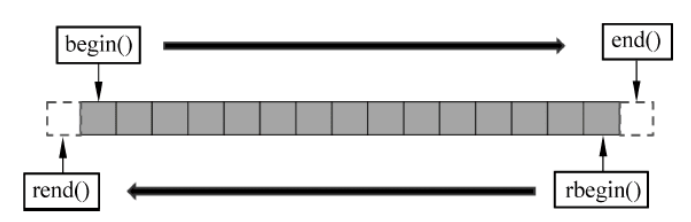


图 13-8 begin()、end()、rbegin()、rend()返回序列的前向和逆向的首尾迭代器

```
#include <vector>
int main() {
    std::vector<int> vec{ 1,2,3,4,5 };
    std::vector<int>::iterator it;
    std::vector<int>::const_iterator cit;
    for (it = vec.begin(); it != vec.end(); it++)
        *it *= 2;                       //可以通过*it 修改它指向的元素的值

    for (cit = vec.cbegin(); cit != vec.cend(); cit++)
        std::cout << *cit << '\t';      //只能查询,不能修改 cit 指向的元素
    std::cout << '\n';

    //const_reverse_iterator 逆向迭代器
    std::vector<int>::const_reverse_iterator crit;
    //crbegin()指向最后一个元素
    for (crit = vec.crbegin(); crit != vec.crend(); crit++)
        std::cout << *crit << '\t';
```

```
        std::cout << '\n';
    }
```

执行程序，输出结果：

```
2       4       6       8       10
10      8       6       4       2
```

和指针一样，如果迭代器指向的是一个类类型对象，可以用->运算符访问这个类的成员（变量或函数），如下面代码的迭代器 it 指向的数据元素是一个 string 对象，可用 it-> size()查询这个 string 对象的字符个数。

```
#include <iostream>
#include <vector>
#include <string>
using std::string;
int main() {
        std::vector<string> vec{ "hello","world","c++","python" };
        std::vector<string>::iterator it = vec.begin() + 2;
        std::cout << it->size() << '\n';     //输出：3
}
```

前面看到，序列容器提供基于迭代器的插入（insert()）和删除操作（erase()）。insert()和 erase()不但可以插入、删除一个元素，还可以插入、删除一个范围的元素。如：

```
int main() {
        std::vector<int> v{ 1,2,3 };
        std::vector<int> vec{ 4,5,6 };
        int arr[]{ 7,8};
        v.insert(v.end(), vec.begin(), vec.end());  //在 v.end()位置插入[vec.begin(), vec.
                                                    //end())之间的元素
        v.insert(v.end(), arr, arr + std::size(arr));  //范围可以是一个指针指向的范围
                                              //[arr, arr + std::size(arr))
                                              //std::size()可以查询一个内在数组的大小
        print(v);       //输出：1     2     3     4     5     6     7     8
        v.erase(v.begin() + 1, v.end() - 2);
        print(v);       //输出：1     7     8
}
```

v.insert(v.end()，vec.begin()，vec.end())表示在 v 的最后一个元素的后一个位置(v.end())处插入 vec.begin()和 vec.end()两个迭代器之间（不包括 vev.end()位置）的所有元素。

基于迭代器的 insert()、erase()、assign()（赋值）的重载函数模板有：

```
//在迭代器 pos 位置前插入一个值 val
iterator insert(iterator pos, const value_type & val);
```

```
//在迭代器 pos 位置前插入 n 个值 val
void insert(iterator pos, size_type n, const value_type & val);

//在迭代器 pos 位置插入[first, last)的值
template < class InputIterator >
void insert(iterator pos, InputIterator first, InputIterator last)

//删除迭代器 pos 位置的元素
iterator erase(iterator pos);

//删除迭代器[first, last)的元素
iterator erase(iterator first, iterator last);

//赋值：将旧内容删除,赋值为 n 个 val 值
void assign(size_type n, const value_type & val);

//用范围 [first, last)的值赋值
template < class InputIterator >
void assign(InputIterator first, InputIterator last);

//用初始化列表中的元素赋值
void assign(initializer_list < value_type > il);
```

例如,可以用 assign()给一个序列容器赋值。

```
#include < iostream >
#include < vector >

void print(const std::vector < int > &v){
    for(auto e:v)   std::cout << e <<"   ";
    std::cout << std::endl;
}

int main(){
    std::vector < int > first;
    std::vector < int > second;
    std::vector < int > third;

    first.assign(7, 100);                //用 7 个 100 赋值
    print(first);

    first.assign({1,2,3,4,5,6,7});
    print(first);


    std::vector < int >::iterator it;
    it = first.begin() + 1;
```

```
    //用[it, first.end() - 1)里即 first 的 5 个中间元素赋值
    second.assign(it, first.end() - 1);
    print(second);

    int myints[] = { 1989,7,4 };
    //用数组范围[myints, myints + 3)里的元素赋值
    third.assign(myints, myints + 3);
    print(third);
}
```

执行程序,输出结果:

```
100  100  100  100  100  100  100
1  2  3  4  5  6  7
2  3  4  5  6
1989  7  4
```

定义迭代器变量写出完整的迭代器类型是麻烦的,应该用 auto 从容器的 begin()等函数返回的迭代器推断迭代器的类型而不是写出复杂的迭代器类型。

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> vec{ 1,2,3 };
    for (auto it = vec.begin(); it != vec.end(); it++)
        *it *= 2;
    for (auto cit = vec.cbegin(); cit != vec.cend(); cit++)
        std::cout << *cit << '\t';
    std::cout << std::endl;              //输出: 2     4     6
}
```

2. 输入迭代器、输出迭代器、前向迭代器、双向迭代器、随机访问迭代器

不同容器的迭代器支持的迭代器操作是有区别的。所有容器的迭代器都支持++、*、->、==和!=运算。其中,++用于前向移动迭代器,指向下一个位置;*运算符用于读或写迭代器指向的元素;如果迭代器指向的是一个类类型的对象,可以用->间接访问迭代器指向对象的成员。==和!=运算符比较 2 个迭代器是否相等或不等。

根据支持的迭代器操作的不同,迭代器可分为 5 种,如表 13-1 所示。

输入迭代器(input iterator):用*运算符可读取迭代器指向元素的值,如 istream 就提供这种迭代器。

输出迭代器(output iterator):用*运算符可给迭代器指向元素赋一个值(即写操作),如 ostream 就提供这种迭代器。

前向迭代器(forward iterator):同时继承 input iterator 和 output iterator,即*运算符可以用于读写迭代器指向的元素,如 forward_list<>、unordered_set<>、unordered_map<>和 unordered_multimap<>容器的迭代器。

双向迭代器(bidirectional iterator):支持迭代器前向和逆向移动,即支持--运算,但

不支持和整数的+=、-=、+和-运算，如 list <>、set <>、map <>、multiset <>、multimap <>容器的迭代器。

随机访问迭代器(random-access iterators)：除双向迭代器的运算外，还支持和整数的算术运算+=、-=、+、-和比较运算<、<=、>及>=，如 vector <>、array <>、deque <>容器的迭代器。另外，string 和内置数组也提供这种迭代器。

表 13-1　迭代器的种类

迭代器形式	描　述
输入迭代器(input iterator)	只读，前向移动
输出迭代器(output iterator)	只写，前向移动
前向迭代器(forward iterator)	读写，前向移动
双向迭代器(bidirectional iterator)	读写，前向和后向移动
随机访问迭代器(random-access iterator)	读写，随机访问

这 5 种迭代器是一个层次继承关系(如图 13-9 所示)，即 random-access iterator 继承自 bidirectional iterator，bidirectional iterator 继承自 forward iterator，forward iterator 继承自 input iterator 和 output iterator。input iterator 和 output iterator 派生自 iterator。

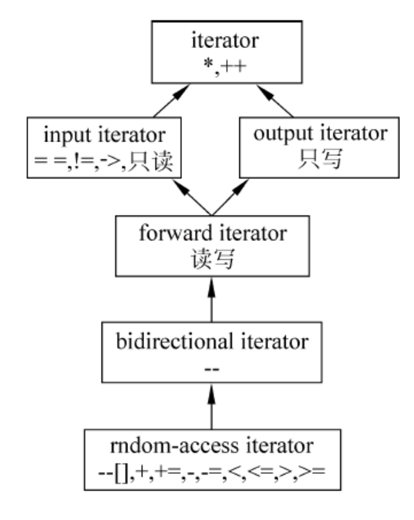


图 13-9　迭代器的继承关系

注意：3 个适配器 std::stack <>、queue <>和 priority_queue <>不提供迭代器。

13.3.2　迭代器适配器

在< iterator >中，标准库提供了下列适配器从给定的迭代器类型生成相关类型的迭代器。

- 逆向迭代器(reverse iterator)：这种迭代器逆向而不是前向移动。具有双向迭代器的库容器都具有逆向迭代器。
- 插入迭代器(insert iterator)：这种迭代器绑定到容器并可用于将元素插入容器中。
- 流迭代器(stream iterator)：这种迭代器绑定到输入或输出流，可用于迭代访问关联的 I/O 流。

- 移动迭代器(move iterator)：这种特殊用途的迭代器用于移动而不是复制它们指向的元素。

1. 逆向迭代器

使用迭代器，可以从 b 到 e 遍历序列[b：e)。如果序列允许双向访问，也可以逆向遍历序列，即从 e 到 b，这样的迭代器就叫作 reverse_iterator。reverse_iterator 从其底层迭代器定义的序列[b,e)的尾部迭代到序列的开头，即[e−1,b−1)。

逆向迭代器(reverse iterator)从其底层序列的末尾开始迭代，其++运算符沿着序列的逆向移动。可以通过容器的 rbegin()、crbegin()获得序列容器的尾部元素的逆向迭代器，而 rend()、crend()获得序列容器的第一个元素的逆向前一个位置的逆向迭代器。

逆向迭代器是一个普通的迭代器，例如下面的程序可以从一个序列容器 C 的尾部开始查找第一个值等于 v 的元素：

```
template<typename C,Val v>
auto find_last(C& c,Val v) ->decltype(c.begin())    //返回一个正向迭代器
{
    for (auto p = c.rbegin(); p != c.rend(); ++p)   //逆向搜索
        if ( *p == v) return --p.base();            //返回一个正向迭代器
    return c.end();                                 //用 c.end()表示"未找到"
}
```

如果用前向迭代器，可以写成下面的等价形式：

```
template<typename C>
auto find_last(C& c, Val v) ->decltype(c.begin())
{
    for (auto p = c.end(); p != c.begin(); )        //前向搜索 search backward from end
        if ( * --p == v) return p;
    return c.end();                                 //用 c.end()表示"未找到"
}
```

注意：逆向迭代器和正向迭代器错开一个位置。

2. 插入迭代器

为了理解为什么需要插入迭代器，先看一个复制的例子。

```
#include<iostream>                                  //std::cout
#include<algorithm>                                 //std::copy
#include<vector>                                    //std::vector
#include<list>                                      //std::list
int main() {
    int arr[] = { 1,2,3 };
    std::list<int> l{ -1, -2, -3, -4 };
    std::copy(arr, arr + 3, l.begin());
    std::cout << "l contains:";
    for (std::list<int>::iterator it = l.begin(); it != l.end(); ++it)
        std::cout << ' '<< * it;
```

```
    std::cout << '\n';
    return 0;
}
```

执行程序，输出结果：

```
l contains: 1 2 3 -4
```

copy()函数模板的规范是：

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result)
```

它将前2个输入迭代器(InputIterator)first和last指定范围的元素复制到输出迭代器(OutputIterator)result的位置，并且覆盖了输出迭代器原来指向的那些元素的值。

如果输出迭代器result指向的容器的范围小于输入迭代器的范围，则会导致非法内存访问，引起程序崩溃。例如下面的代码：

```
int arr[] = { 1,2,3,4,5};
std::list<int> l{ -1, -2, -3, -4 };
std::copy(arr, arr + 5, l.begin());
```

因为copy()的范围[arr,arr + 5)有5个元素，而l只有4个元素，这段代码会引起程序崩溃。

插入迭代器std::**insert_iterator**是一个迭代器适配器，它接收一个容器，产生一个迭代器，能实现向给定容器**插入元素而不是覆盖**已有的元素，并且迭代器的位置会自动前移。插入迭代器实际上是调用了容器内部的insert()成员函数执行插入操作的。

```
#include <vector>
#include <list>
#include <iostream>
#include <iterator>
#include <algorithm>
int main(){
    std::vector<int> v{ 1,2,3,4,5 };
    std::list<int> l{ -1, -2, -3 };
    std::copy(v.begin(), v.end(),
        std::insert_iterator<std::list<int>>(l,l.begin()));
    for (int n : l)
        std::cout << n << ' ';
    std::cout << '\n';
}
```

上述代码中std::**insert_iterator**<std::list<int>>(l,l.begin())创建了一个std::list<int>类型的插入迭代器。可以看到插入迭代器的构造函数的第1个参数是容器对象，第2个参数是该容器的一个迭代器。std::copy将[v.begin(),v.end())的元素复制到这个插入

迭代器位置处，并没有覆盖掉 l 的原有内容。

执行程序，输出结果：

```
1 2 3 4 5 -1 -2 -3
```

可以用辅助函数 std::inserter()简化插入迭代器的使用。

```
std::copy(v.begin(), v.end(), std::inserter(l, l.begin()));
```

除了 std::insert_iterator 外，还有 2 个分别在容器的头部和尾部插入元素的首端插入迭代器(front_insert_iterator)和尾端插入迭代器(back_insert_iterator)。std::**front_insert_iterator** 使用 push_front()在序列的第一个元素之前插入。std::**back_insert_iterator** 使用 push_back()在序列的最后一个元素之后插入。

它们对应的辅助函数模板 std::front_inserter 和 std::back_inserter 可以简化这 2 个插入迭代器对象的构造。和 std::inserter 需要第 2 个参数传递一个迭代器不同，std::front_inserter 和 std::back_inserter 只需要传递一个容器对象就可以。

```
std::inserter(c, p)                         //c 是容器对象,p 是指向 c 的迭代器
std::front_insert_iterator(c)               //c 是容器对象
std::back_insert_iterator(c)                //c 是容器对象
```

当然也可以直接用插入迭代器自身的构造函数定义插入迭代器对象，如：

```
#include <vector>
#include <iostream>
#include <iterator>
using namespace std;

void Print(const vector<int>& v) {
    for (auto e:v)              cout << e << '\t';
    cout << endl;
}
int main() {
    vector<int> v{1,2,3};
    insert_iterator<vector<int>> p{ v,v.begin() + 1 };
    back_insert_iterator<vector<int>> bp{v};
    *p++ = 20;                                  //*p = 20 然后 p++
    *p = 10;        Print(v);
    * bp = 30;    Print(v);

    //错: vector 没有 push_front()
    //front_insert_iterator<vector<int>> fp{ v };
    //* fp = 40;    Print(v);
}
```

注意：vector 不支持 push_front()，因此不能使用 front_insert_iterator。

3. 流迭代器

流迭代器(stream iterator)使得输入输出流可以被当作一个序列容器一样访问。其中 istream_iterator 用于读取输入流，而 ostream_iterator 用于向输出流写数据。

当创建一个输入流迭代器 std::istream_iterator 对象时，可以指定一个输入流对象，如果不指定一个输入流对象，那么这个迭代器对象就是一个**尾迭代器对象**(指向输入流的最后一个元素的后一个位置)，相当于空指针。例如：

```
#include <iostream>
#include <iterator>
int main() {
    std::istream_iterator<int> it(std::cin);    //创建一个 istream_iterator<int>类型的
                                                //流迭代器对象 it
    std::istream_iterator<int> eof;             //尾迭代器，相当于空指针
    while (it != eof)                           //当迭代器未到达尾迭代器位置时，就循环
        std::cout << *it++;
}
```

上述程序创建了数据类型是 int 的 istream_iterator 流迭代器对象 it 和 eof，其中 eof 表示输入流的结束位置。程序循环从该迭代器读取 int 类型数，直到遇到不是 int 类型的数据或文件结束符。下面是执行程序的情况：

```
1 2 3 hello
123
```

下面的代码构造了读取文件输入流对象 iF 的 istream_iterator<string>迭代器 it 和 eof，因此，循环过程中，it 指向的是一个字符串。

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <string>
int main() {
    std::ifstream iF("test.txt");
    std::istream_iterator<std::string> it(iF);  //从 iF 读取数据
    std::istream_iterator<std::string> eof;     //尾迭代器
    while (it != eof)
        std::cout << *it++ << '\t';
}
```

执行程序，输出结果：

```
B站和微博用户名:	hw-dong 博客网址:	https://a.hwdong.com	哈罗 hello$ %346dsfsd@#
```

创建一个输出流迭代器 std::ostream_iterator 对象，必须绑定一个输出流对象，还可以传递第 2 个参数，这个参数是一个 C 风格字符串，表示输出每个元素后会输出这个参数表

示的字符串。通过解引用运算符 * 给输出流迭代器赋值，就是将这个值输出到其关联的那个输出流对象中。例如：

```
#include <iostream>
#include <iterator>
int main() {
    int arr[] = { 1,3,5 };
    std::ostream_iterator<int> out_iter(std::cout, "- *");
    for (auto e : arr)
        *out_iter++ = e;                    //给输出流迭代器赋值就是将
                                            //该元素输出到其关键的输出流对象中
    }
```

执行程序，输出结果：

```
1- *3- *5- *
```

4. 移动迭代器

移动迭代器也是一个迭代器适配器，使用解引用运算符 * 作用于它并赋值给其他变量时，执行的是移动而不是复制。假如 it 是一个移动迭代器，执行 x = *it 是将 *it 的内容移动到 x 中而不是赋值到 x 中，即执行的是移动语义，从而提高了程序的效率。

通常使用辅助函数 make_move_iterator()从另一个迭代器中创建一个移动迭代器，例如：

```
mp = make_move_iterator(p)
```

上述代码从迭代器 p 创建了一个移动迭代器 mp。

下面的程序用 std::uninitialized_copy(arr,arr+n,tmp)将 arr～arr+n 的数据复制到 temp 指向的更大内存空间。

```
#include <memory>
#include <iostream>
class X {
    int a{ 0 };
public:
    X() = default;
    int get() { return a; }
    void set(const int a) { this->a = a; }
    X(const X& x) :a{ x.a } {
        std::cout << "复制数据:" << a << "\n"; }
    X(const X&& x) :a{ x.a } {
        std::cout << "移动数据:" << a << "\n";
    }
};
int main(){
```

```
    int n{5};
    X * arr = new X[n];
    for (auto i{ 0 }; i != n; i++)
        arr[i].set(2 * i + 1);
    //分配更大的内存块
    auto n2{ 2 * n };
    X * tmp = new X[n2];
    //将 arr 数据复制到这个更大的内存块中
    auto last = std::uninitialized_copy(arr, arr + n, tmp);
    delete[] arr;                                  //释放原来 arr 指向的内存
    arr = tmp;                                     //让 arr 指向这个新的内存块
    //输出新内存块的数据
    for (auto i{ 0 }; i != n; i++)
        std::cout << arr[i].get()<<" ";
}
```

arr 先指向了 5 个 X 元素的一块内存，后面假如需要更大的内存，就有分配了 10 个 X 元素的空间，然后用 std::uninitialized_copy()将 arr 指向的旧空间(即范围[arr,arr+n))内容复制到 tmp 指向的新空间里，默认是对每个元素都执行了复制操作，然后释放旧空间(delete[] arr)，并将 arr 指向新空间。

执行程序，输出结果：

```
复制数据:1
复制数据:3
复制数据:5
复制数据:7
复制数据:9
1 3 5 7 9
```

由于旧空间内容不再需要，可以用移动语义直接将旧空间的内容移动到新的空间中。

```
auto last = std::uninitialized_copy(std::make_move_iterator(arr),
            std::make_move_iterator(arr + n), tmp);
```

当用上句代码替换之前的 std::uninitialized_copy()，执行的就是移动而不是复制。

执行程序，输出结果：

```
移动数据:1
移动数据:3
移动数据:5
移动数据:7
移动数据:9
1 3 5 7 9
```

这里的 X 类仅仅是为了说明是执行了复制还是移动，其移动和拷贝构造函数是完全一样的。但如果 X 是一个包含资源的类(如 X 是 string 或 vector)，移动要比复制效率高。

13.3.3 数组、字符串和迭代器

内在数组类型也支持迭代器操作，如可以通过 std::begin()和 std::end()得到一个内置数组的开头和结束位置迭代器，实际返回的迭代器就是指针：

```
#include <iostream>
int main() {
      int arr[]{ 1,2,3,4};
      for (auto it = std::begin(arr); it != std::end(arr); it++)
            std::cout << *it << '\t';
}
```

执行程序，输出结果：

```
1       2       3       4
```

同样，C++的 string 类型也支持迭代器操作，如可以通过 string 的 begin()和 end()成员函数获得一个 string 对象的开头和结束位置迭代器：

```
#include <string>
#include <iostream>
int main() {
    std::string::iterator it;
    std::string str = "hello world";
    for (it = str.begin(); it < str.end(); it++)
        std::cout << *it;
    std::cout << std::endl;
}
```

13.4 算法

和容器、迭代器一样，算法是 C++标准库的重要组成部分。在 C++头文件<algorithm>中定义了大量(C++17 有 105 个)以函数模板形式存在的标准算法，可以用于搜索、排序、计数等各种操作。这些函数模板结合了头等函数、高阶函数、迭代器等技术，可以对一对迭代器指定范围的序列或一个迭代器指示位置的元素进行处理。当复制或比较 2 个序列时，第 1 个序列由一对迭代器如[b,e)表示，第 2 个序列由一个迭代器如 b2 表示其起始位置，那么第 2 个序列范围就是[b2,b2＋e－b)。大多数算法如 find()只需要前向迭代器，也有一些算法如 sort()需要的是随机迭代器。

容器的成员函数只能对容器类型的对象进行操作，而基于迭代器的算法可以用于不同的容器对象，只要该容器提供了相应的迭代器(如前向迭代器或随机迭代器)。因此，标准库的基于迭代器的这些算法比容器自身的成员函数更灵活、更具有通用性。

基于迭代器，程序员也可以自己实现各种通用算法。

13.4.1 自定义通用算法

例如，前面的函数模板：

```
template<typename T, typename CompareT>
    T find_optimal(const T * arr, const int n, CompareT compare)
```

该模板的前 2 个参数就是一个数组类型的指针和大小，因此这个函数虽然可以用于任何类型的数组，但不能用于其他的集合（容器）类型，如标准库的 vector、list 类型。可以用迭代器指定序列的范围[first，last)，即将 find_optimal 改写为如下模板：

```
template<typename Iterator, typename CompareT>
Iterator find_optimal(Iterator begin, Iterator end, CompareT compare) {
    if (begin == end) return end;
    Iterator optimum = begin;
    for (Iterator iter = ++begin; iter != end; ++iter)
    {
        if (compare( * iter, * optimum))
        {
            optimum = iter;
        }
    }
    return optimum;
}
```

该模板添加了模板参数 typename Iterator 用于表示迭代器类型，对于函数的形参现在就可以用 2 个迭代器对象表示序列的范围([begin，end))，第 3 个函数形参仍然是比较 2 个对象的函数。这个函数模板就不仅仅是作用于数组而是作用于任何可以提供序列迭代器的容器对象，因此，更具有通用性。

```
#include<iostream>
int main() {
    std::vector<int> arr{ 3,11,51,25,7,39,68 };

    std::cout << "输入一个数值: \n";
    double v;  std::cin >> v;
    std::cout << * find_optimal(arr.begin(),arr.end(), [=](double x, double y) {
        return std::abs(x - v) < std::abs(y - v); }) << '\t';

    std::cout << * find_optimal(arr.begin() + 2, arr.end() - 2, [=](double x, double y) {
        return std::abs(x - v) < std::abs(y - v); }) << '\t';

    int arr2[]{ 19,3,24,42,53,26 };
    std::cout << * find_optimal(std::begin(arr2) + 1, std::end(arr2) - 1, [=](double x,
double y) {
        return std::abs(x - v) < std::abs(y - v); }) << '\t';
}
```

可以看到该函数模板不但可以用于 std::vector<>对象(如 arr),也可以用于普通的数组(如 arr2)。不管是 arr.begin()和 arr.end()返回的迭代器,还是 std::begin(arr2)和 std::end(arr2)的普通指针,只要能用 * 运算符访问这个指示器指向的元素,就能应用这个函数模板。

C++标准库的算法都是这种基于迭代器的函数模板,使得算法可以用于任何集合(容器)类型的对象。这样的算法也称为**泛型算法**。

13.4.2 策略参数

许多标准库算法除了用迭代器表示序列范围或某个位置外,还可以接收一个表示算法策略的参数。例如上面的 find_optimal()的第 3 个参数 compare 就是一个策略参数,其对应的类型模板参数是 CompareT。给这个策略参数传递不同的模板类型实参(不同的头等函数)就能让 find_optimal()按照这个 compare 参数的比较含义对 2 个元素进行比较。上述代码中传递了一个 Lambda 函数作为这个策略参数,用这个 Lambda 函数对 2 个元素比较大小。

标准库的求最小值、最大值的算法分别是 std:: min_element()、std:: max_element():

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    std::vector<int> arr{ 3,11,51,25,7,39,68 };

    std::cout << "\n 最大值是: ";
    std::cout << * std::max_element(arr.begin(), arr.end());

    std::cout << "\n 最小值: ";
    std::cout << * std::min_element(arr.begin(), arr.end());

    int arr2[]{ 19,3,24,42,53,26 };
    std::cout << "\n 最小值: ";
    std::cout << * std::min_element(std::begin(arr2), std::end(arr2));
}
```

算法 std::minmax_element()可以同时求出最小值和最大值,返回的是一个 std::pair<>对象:

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    int arr[]{ 19,3,24,42,53,26 };
    auto [min_it,max_it] = std::minmax_element(std::begin(arr), std::end(arr));
    std::cout << "最小值、最大值分别为: " << * min_it << '\t' << * max_it << '\n';;

    double v;  std::cin >> v;
```

```
    auto[near_it, far_it] = std::minmax_element(std::begin(arr), std::end(arr),
        [v](const auto &x, const auto &y) {return std::abs(x - v) < std::abs(y - v); });

    std::cout << "最小值、最大值分别为: " << *near_it << '\t'<< *far_it << '\n';
}
```

上述代码第一个 std::minmax_element()调用返回的是数组中的最小值、最大值的位置，后一个调用传递了一个 Lambda 函数作为策略参数，用这个 Lambda 函数对 2 个数据元素进行比较，即比较 2 个元素哪个距离捕获参数 v 更近。std::minmax_element()将返回距离输入的值 v 最近或最远的元素的位置。

上述 3 个求最值的算法都可以传递一个比较 2 个对象的头等函数(如这里的 Lambda 表达式)作为策略。

13.4.3 标准库的常用算法

STL 的头文件<algorithm>中包含了很多算法。这些算法可以按照是否修改(容器)序列而分为查询型算法和修改型算法。查询型算法(如 find()、search()等)只是读取元素的值，不会修改元素的值或改变元素的排列次序。修改型算法(如 for_each()、sort()、replace()、remove()等)会修改序列，如改变排列次序或修改元素的值。

下面是一些常用算法。

1. 通用迭代：for_each()

for_each()用于迭代地对序列中的每个元素执行某个操作。

for_each(b,e,f)：将一个操作 f 应用到序列[b:e)中的每个元素。如：

```
void square_all(vector<int>& v) {                  //对 v 的每个元素执行平方运算
    for_each(v.begin(), v.end(), [](int& x) {x *= x; });
}
```

再如：

```
#include <vector>
#include <algorithm>
#include <iostream>

struct Sum{
    Sum() : sum{ 0 } { }
    void operator()(int n) { sum += n; }
    int sum;
};

int main(){
    std::vector<int> nums{ 3, 4, 2, 8, 15, 267 };
    auto print = [](const int& n) { std::cout << " " << n; };
    std::cout << "用 print 输出所有数:";
    std::for_each(nums.begin(), nums.end(), print);
```

```
    std::cout << '\n';
    //对每个数执行 Lambda 表达式,即每个数增加 1
    std::for_each(nums.begin(), nums.end(), [](int &n) { n++; });
    //对每个数执行函数调用 Sum::operator()
    Sum s = std::for_each(nums.begin(), nums.end(), Sum());
    std::cout << "after: ";
    std::for_each(nums.begin(), nums.end(), print);
    std::cout << '\n';
    std::cout << "sum: " << s.sum << '\n';
}
```

2. 计数：count()

count()计数匹配的或满足某个条件的元素个数。下面是 6 个函数模板中的 2 个。

count(b,e,x)：序列中等于 x 的元素个数。

count_if(b,e,f)：序列中元素 x 满足 f(x)的元素个数。

例如,下面代码统计 3 出现的次数和被 3 整除的数的个数：

```
vector<int> v{ 1, 2, 3, 4, 4, 3, 7, 8, 9, 10 };
cout << count(v.begin(), v.end(), 3)<< '\t';
cout << count_if(v.begin(), v.end(), [](int i) {return i % 3 == 0; });
```

3. 查找：find()

find()用于查找满足条件的元素。有如下版本的一些 find()函数模板。

p=find(b,e,v)：p 指向[b:e)中满足 *p==v 的第一个元素。

p=find_if(b,e,f)：p 指向[b:e)中 f(*p)为 true 的第一个元素。

p=find_if_not(b,e,f)：p 指向[b:e)中 f(*p)为 false 的第一个元素。

p=find_first_of(b,e,b2,e2)：p 指向[b:e)中满足 *p== *q(q 是[b2:e2)的某个元素)的第一个元素。

p=find_first_of(b,e,b2,e2,f)：p 指向[b:e)中 f(*p, *q)(q 是[b2:e2)的某个元素)为 true 的第一个元素。

p=adjacent_find(b,e)：p 指向[b:e)中满足 *p== *(p+1)的第一个元素。

p=adjacent_find(b,e,f)：p 指向[b:e)中满足 f(*p, *(p+1))为 true 的第一个元素。

p=find_end(b,e,b2,e2)：p 指向[b:e)中满足 *p== *q(q 是[b2:e2)的某个元素)的最后一个元素。

p=find_end(b,e,b2,e2,f)：p 指向[b:e)中 f(*p, *q)(q 是[b2:e2)的某个元素)为 true 的最后一个元素。

例如：

```
#include <vector>
#include <algorithm>
#include <iostream>
int main() {
    std::vector<int> v{ 3,53,1,19,24,42, 3,26 };
```

```
    std::cout << "3 出现的次数: " << std::count(v.begin(), v.end(), 3) << '\n';
    auto it = std::find(v.begin(), v.end(), 3);  //寻找等于 3 的第一个元素
    if (it != v.end()) std::cout << * it << '\t';
    auto x{ 6 };
    //第一个大于 x 的元素
    it = std::find_if(v.begin(), v.end(), [x](const auto &a) {return a > x; });
    if (it != v.end()) std::cout << * it << '\t';
    //第一个不大于 x 的元素
    it = std::find_if_not(v.begin(), v.end(), [x](const auto &a) {return a > x; });
    if (it != v.end()) std::cout << * it << '\t';
}
```

执行程序，输出结果：

```
3 出现的次数: 2
3       53      3
```

再如：

```
int main(){
    std::vector < int > v{ 1,3,7,4 };
    std::vector < int > t{ 0,2,3,4,5 };

    auto p = std::find_first_of(v.begin(), v.end(), t.begin(), t.end());
    std::cout <<"第一个位于" << std::distance(v.begin(), p) <<"的元素"
        << *p << "在 t 中找到了匹配元素\n";
    auto q = std::find_first_of(p + 1, v.end(), t.begin(), t.end());
    std::cout << "第一个位于" << std::distance(v.begin(), q) << "的元素"
        << *q << "在 t 中找到了匹配元素\n";
}
```

执行程序，输出结果：

```
第一个位于 1 的元素 3 在 t 中找到了匹配元素
第一个位于 3 的元素 4 在 t 中找到了匹配元素
```

4. 搜索：search()

search()在一个序列中搜索等于某个序列的子序列，并返回该子序列的位置。

有 search()和 search_n()共 2 种函数。

p=search(b,e,b2,e2)：p 指向[b:e)中使得[p:p+(e−b))等于[b2:e2)的第一个元素。

p=search(b,e,b2,e2,f)：p 指向[b:e)中使得在 f()作为比较函数的情况下：[p:p+(e−b))等于[b2:e2)的第一个元素。

p=search_n(b,e,n,v)：p 指向[b:e)中使得[p:p+n)具有值 v 的第一个元素。

p=search_n(b,e,n,v,f)：p 指向[b:e)中使得[p:p+n)的每个元素 *q 满足 f(*q,v)为 true 的第一个元素。

例如，下列程序判断字符序列中是否存在4个或3个连续的字符'0'。

```
#include <iostream>
#include <algorithm>
#include <iterator>
template <class Container, class Size, class T>
bool consecutive_values(const Container& c, Size count, const T& v){
    return std::search_n(std::begin(c), std::end(c), count, v) != std::end(c);
}
int main(){
    const char sequence[] = "1001010100010101001010101";
    std::cout << std::boolalpha;
    std::cout << "Has 4 consecutive zeros: "
        << consecutive_values(sequence, 4, '0') << '\n';
    std::cout << "Has 3 consecutive zeros: "
        << consecutive_values(sequence, 3, '0') << '\n';
}
```

执行程序，输出结果：

```
有 4 个连续的 0: false
有 3 个连续的 0: true
```

binary_search(b,e,x)：二分搜索法查找范围[b,e)中是否有等于x的元素。返回true或false。

binary_search(b,e,x,f)：二分搜索法查找范围[b,e)中是否有(按照比较谓词f比较相等)等于x的元素。返回true或false。

lower_bound(b,e,x)：返回[b:e)中第一个不小于x的元素的迭代器位置。

lower_bound(b,e,x,f)：返回[b:e)中第一个按照比较谓词f不小于x的元素的迭代器位置。

upper_bound(b,e,x)：返回[b:e)中第一个大于x的元素的迭代器位置。

upper_bound(b,e,x,f)：返回[b:e)中第一个按照比较谓词f大于x的元素的迭代器位置。

例如：

```
#include <vector>
#include <algorithm>
#include <iostream>
int main() {
    std::vector<int> v{10, 20, 30, 30, 20, 10, 10, 20};
    std::sort(v.begin(), v.end());
    for(auto e:v)
       std::cout << e <<" ";
    std::cout <<'\n';
    if (binary_search(v.begin(), v.end(), 20))
        std::cout << "存在等于 20 的元素\n";
    auto low = std::lower_bound(v.begin(), v.end(), 20);
```

```
    auto up = std::upper_bound(v.begin(), v.end(), 20);
    std::cout << low - v.begin()<<'\t'<< *low << '\n'
        << up - v.begin()<<'\t'<< *up << std::endl;
}
```

执行程序，输出结果：

```
10 10 10 20 20 20 30 30
存在等于 20 的元素
3       20
6       30
```

5．序列谓词（断言）：all_of()、any_of()、none_of()

all_of()、any_of()、none_of()判断[b,e)范围中的一个或全部是否满足或不满足某个谓词。

all_of(b,e,f)：判断对[b:e)中的所有元素 x,f(x)是否都返回 true。

any_of(b,e,f)：判断[b:e)中是否存在元素 x,使得 f(x)返回 true。

none_of(b,e,f)：判断对[b:e)中的所有元素 x,f(x)是否都返回 false。

例如，可以用 all_of()判断一个 vector 对象中的数是否都是偶数：

```
std::vector<int> v(10, 2);
if (std::all_of(v.cbegin(), v.cend(), [](int i) { return i % 2 == 0; })) {
    std::cout << "All numbers are even\n";
}
```

这 3 个算法实际可以用前面的 find_if 模板来实现。

```
template<class InputIt, class UnaryPredicate>
bool all_of(InputIt first, InputIt last, UnaryPredicate p){
    return std::find_if_not(first, last, p) == last;
}
template<class InputIt, class UnaryPredicate>
bool any_of(InputIt first, InputIt last, UnaryPredicate p){
      return std::find_if(first, last, p) != last;
}
template<class InputIt, class UnaryPredicate>
bool none_of(InputIt first, InputIt last, UnaryPredicate p){
      return std::find_if(first, last, p) == last;
}
```

6．比较：相等 equal()和不匹配 mismatch()

equal()比较一对序列是否相等。mismatch()比较一对序列是否不匹配并返回第一个不匹配的元素的位置。

equal(b,e,b2)：是否[b:e)和[b2:b2+(e−b))2 个序列的所有对应元素都相等。

equal(b,e,b2,f)：是否[b:e)和[b2:b2+(e−b))2 个序列的所有对应元素 v 和 v2 的

f(v,v2)都是 true。

pair(p1,p2)=mismatch(b,e,b2)：p1 和 p2 分别是[b:e)和[b2:b2+(e−b))中不满足 *p1== *p2 的第一个元素，或者 p1 等于 e。

pair(p1,p2)=mismatch(b,e,b2,f)：p1 和 p2 分别是[b:e)和[b2:b2+(e−b))中 f(*p1, *p2)为 false 的第一个元素，或者 p1 等于 e。

例如，如果一个字符串和其逆向字符串完全一样，该字符串称为**回文**(palindrome)。下面代码求一个字符串中回文的最长子串。

```
#include <iostream>
#include <string>
#include <algorithm>

std::string mirror_ends(const std::string& in){
    return std::string(in.begin(),
        std::mismatch(in.begin(), in.end(), in.rbegin()).first);
}

int main(){
std::cout << mirror_ends("abXYZba") << '\n'
    << mirror_ends("abca") << '\n'
    << mirror_ends("aba") << '\n';
}
```

执行程序，输出结果：

```
ab
a
aba
```

注：std::**pair** 是一个 struct 模板，它提供了将异质对象存储为单个单元的方法。一个 pair 对象是具有 2 个元素的 std::**tuple** 的特定情况。类模板 std::tuple 是固定大小的多个异质对象的集合。通常用 std::make_pair 或 std::make_tuple 构造一个 pair 或 tuple 对象。

对于 pair 对象，通过其 first 和 second 两个属性访问其包含的 2 个异质对象。例如：

```
#include <string>
#include <iostream>>
int main() {
    std::pair<std::string, double> name_score("张伟",89.5);
    name_score.second = 90.5;
    std::cout << name_score.first << '\t'<< name_score.second << '\n';
    auto p = std::make_pair("赵四", 70.5);
    std::cout << p.first << '\t'<< p.second << '\n';
}
```

执行程序，输出结果：

```
张伟    90.5
```

```
赵四    70.5
```

对于 tuple 对象，可以用 std::get <>传递一个索引来访问其中的数据元素，也可以用 std::tie 解开其元素到单独变量中。例如：

```
#include <string>
#include <iostream>>
std::tuple<std::string,double, char> get_student() {
    std::string name; double score; char level;
    std::cin >> name >> score >> level;
    return std::make_tuple(name,score, level);
}

int main() {
    std::string name; double score; char level;
    std::tie(name, score, level) = get_student();
    std::cout << name << '\t'<< score << '\t'<< level << '\n';
    auto s = get_student();
    std::cout << std::get<0>(s) << '\t'<< std::get<1>(s)
        << '\t'<< std::get<2>(s) << '\n';
}
```

执行程序，输出结果：

```
李平    60.5    E
李平    60.5    E
张伟    90.5    A
张伟    90.5    A
```

7. 排序 sort()和逆置 reverse()

sort(b,e)：对[b:e)的元素进行排序，可以传入比较 2 个元素大小的头等函数 compare。

reverse(b,e)：将一个序列按逆序排列。

例如：

```
#include <algorithm>
#include <functional>
#include <array>
#include <iostream>

template<typename C>
void print(const C& c) {
    for(auto e:c)        std::cout << e << " ";
    std::cout << '\n';
}
int main(){
    std::array<int, 5> s = { 5, -7, 4, -2, 8 };
    //逆序排序
```

```
    std::reverse(s.begin(), s.end());           //将 s 逆置为: 8, -2, 4, -7, 5
    print(s);
    //使用默认的运算符 operator<
    std::sort(s.begin(), s.end());              //默认元素的大小排序
    print(s);
    //使用标准库的比较函数对象
    std::sort(s.begin(), s.end(), std::greater<int>());  //用策略 std::greater 排序
    print(s);
    //使用定制的函数对象
    struct {
        bool operator()(int a, int b) const
        {
            return a < b;
        }
    } customLess;
    std::sort(s.begin(), s.end(), customLess);  //用策略 customLess 排序
    print(s);
    //用 Lambda 表达式比较
    std::sort(s.begin(), s.end(), [](int a, int b) {return a > b;   }); //用 Lambda 策略
                                                                         //排序
    print(s);
}
```

执行程序,输出结果:

```
 8  -2  4  -7   5
-7  -2  4   5   8
 8   5  4  -2  -7
-7  -2  4   5   8
 8   5  4  -2  -7
```

8. 累加: accumulate()

accumulate(b,e,initial_value,BinaryOperation op):对[b:e)中的元素求累加和,累加和有一个初始值 initial_value,op 表示对 2 个元素相加的操作。该函数定义在头文件<numeric>中,对所有的元素累加(或执行 op 操作)。

```
#include <iostream>
#include <vector>
#include <numeric>
#include <string>
#include <functional>

int main() {
    std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int sum = std::accumulate(v.begin(), v.end(), 0);
    int product = std::accumulate(v.begin(), v.end(), 1, std::multiplies<int>());

    std::string s = std::accumulate(std::next(v.begin()), v.end(),
```

```
        std::to_string(v[0]),                    //初始值
        [](std::string a, int b) {               //"相加"操作的定义为这个 Lambda 表达式
        return a + '-' + std::to_string(b);
    });

    std::cout << "和: " << sum << '\n'
        << "积: " << product << '\n'
        << "-分隔的字符串: " << s << '\n';
}
```

执行程序，输出结果：

```
和: 55
积: 3628800
-分隔的字符串: 1-2-3-4-5-6-7-8-9-10
```

9. 变换：transform()

transform()对一个序列中的每个元素执行一个变换操作，将结果输出到另一个序列中。

p=transform(b,e,out,f)：对[b:e)中的每个元素 *p 执行变换操作，将结果写到输出序列[out:out+(e−b))的对应元素 *q 上，即 *q=f(*p)。返回 p=out+(e−b)。

p=transform(b,e,b2,out,f)：对[b:e)中的每个元素 *p 和[b2:b2+(e−b))的对应元素 *q 执行变换操作 *q=f(*p, *q)，将结果写到 out 序列[out:out+(e−b))的对应元素 *r 上。

第一个 transform()函数模板的实现大致如下。

```
template<class In, class Out, class Op>
Out transform(In first, In last, Out res, Op op){
    while (first != last)
        * res++ = op( * first++);
    return res;
}
```

输出序列和输入序列可以是同一个序列，例如下列函数将输入字符串的字母都改为大写字母。

```
void toupper(string& s) {
  transform(s.begin(), s.end(), s.begin(), toupper);
}
```

10. 复制：copy()

copy()将一个序列中的元素复制到另一个序列中。

p=copy(b,e,out)：将[b:e)中的所有元素复制到[out:p)中。p=out+(e−b)。

p=copy_if(b,e,out,f)：将[b:e)中的元素 x 的 f(x)复制到序列[out:p)中。p=out+(e−b)。

p=copy_n(b,n,out)：将[b:b+n)中的前 n 个元素复制到[out:p)中。p=out+n。

p=copy_backward(b,e,out)：从最后一个元素开始，将[b:e)中的所有元素复制到[out:p)中。p=out+(e−b)。

p=move(b,e,out)：将[b:e)中的所有元素移动到[out:p)中。p=out+(e−b)。

p=move_backward(b,e,out)：从最后一个元素开始，将[b:e)中的所有元素移动到[out:p)中。p=out+(e−b)。

例如，下面代码将 ld 中大于 n 的数输出到输出流对象 os 中：

```
void f(list<int>&ld, int n, ostream& os){
    copy_if(ld.begin(), ld.end(), ostream_iterator<int>(os),
        [](int x) { return x>n); });
}
```

11. 去重：unique()

unique()删除序列中邻接的重复元素。重复的含义由函数 f(*p, *(p+1))定义。

p = unique(b,e)：删除[b:e)中的邻接的重复元素，使得[b:p)中有相邻的重复项。

p = unique(b,e,f)：删除[b:e)中的邻接的重复元素，使得[b:p)中没有相邻的重复项。

p = unique_copy(b,e,out)：将[b:e)中的元素复制到[out:p)中，不复制相邻的重复项。

p = unique_copy(b,e,out,f)：将[b:e)中的元素复制到[out:p)中，不复制相邻的重复项。

unique 实际上是修改序列中元素的值而没有修改容器，通过修改值，将相邻重复的元素放到了序列的尾部，返回的 p 就是前面的不重复元素序列的后一个位置。可以通过下列代码来验证这一点：

```
int main(){
    std::string s = "abbcccde";
    auto p = unique(s.begin(), s.end());
    std::cout << s <<'\t'<< p - s.begin() << '\t'
        << s.substr(0, p - s.begin())<< '\n';
}
```

执行程序，输出结果：

```
abcdecde        5       abcde
```

可以采用 2 种方法得到一个不重复元素的序列。

(1) 对结果序列容器使用 erase()等操作删除其后面的元素。

(2) 使用 unique_copy()将不重复元素复制到新的容器中。

下面代码采用的是第 2 种方法。

```
string s = "abbcccde";
```

```
string s2;
auto q = unique_copy(s.begin(), s.end(), std::back_inserter(s2),
    [](char c1, char c2) { return c1 == c2; });
cout << s2  << '\n';
```

执行代码,输出结果:

```
abcde
```

12. 删除：remove()

remove()将元素移除到序列的尾部。

p = remove(b,e,v)：从[b:e)中删除值为 v 的元素,使得[b:p)成为(*q ==v)为 false 的元素。

p = remove_if(b,e,v,f)：从[b:e)中删除元素 *q,使得[b:p]成为 f(*q)为 false 的元素。

p = remove_copy(b,e,out,v)：将[b:e)中使得(*q == v)为 false 的元素 *q 复制到[out:p]中。

p = remove_copy_if(b,e,out,f)：将[b:e)中使得 f(*q)为 false 的元素 *q 复制到[out:p]中。

例如：

```
#include <algorithm>
#include <iterator>
#include <string>
#include <iostream>
#include <cctype>
int main(){
    std::string str = "Text with some   spaces";
    std::cout << "before: " << str << "\n";

    std::cout << "after:  ";
    std::remove_copy(str.begin(), str.end(),
        std::ostream_iterator<char>(std::cout), ' ');
    std::cout << '\n';

    std::string str1 = "Text with some   spaces";
    str1.erase(std::remove(str1.begin(), str1.end(), ' '),
        str1.end());
    std::cout << "remove:  " << str1 << '\n';

    std::string str2 = "Text with some   spaces";
    str2.erase(std::remove_if(str2.begin(),
        str2.end(),
        [](unsigned char x) {return std::isspace(x); }),
        str2.end());
```

```
    std::cout << "remove_if:  " << str2 << '\n';
}
```

执行程序，输出结果：

```
before: Text with some   spaces
after:  Textwithsomespaces
remove:  Textwithsomespaces
remove_if:  Textwithsomespaces
```

13. 替换：replace()

replace()用新值替换满足某种条件的元素。

replace(b,e,v,v2)：将[b:e)中满足 *p == v 的元素 *p，替换为 v2。

replace_if(b,e,f,v2)：将[b:e)中 f(*p)为 true 的元素 *p，替换为 v2。

p = replace_copy(b,e,out,v,v2)：将[b:e)中满足 *p == v 的元素 *p，用 v2 替换 *p 的值并复制到 out 指向的序列中。

p = replace_copy_if(b,e,out,f,v2)：将[b:e)中 f(*p,v)为 true 的元素 *p，用 v2 替换 *p 的值并复制到 out 指向的序列中。

例如，下列代码将字符串中的所有的'o'替换为'X'：

```
std::string s = "hello world";
std::replace(s.begin(), s.end(), 'o', 'X');        //所有的'o'替换为'X'
    std::cout << s;
```

而下列代码将所有小于 5 的数替换为 55：

```
std::array<int, 10> s{ 5, 7, 4, 2, 8, 6, 1, 9, 0, 3 };
std::replace_if(s.begin(), s.end(),
        std::bind(std::less<int>(), std::placeholders::_1, 5), 55);
```

14. 填充：fill()

fill()用于赋值和初始化一个序列。

fill(b,e,v)：将 v 赋值给[b:e)中的每个元素。

p = fill_n(b,n,v)：将 v 赋值给[b:b+n)中的每个元素。p=b+n。

generate(b,e,f)：将 f()赋值给[b:e)中的每个元素。

p = generate_n(b,n,f)：将 f()分配给[b:b+n)的每个元素。p=b+n。

uninitialized_fill(b,e,v)：用 v 初始化[b:e)中的每个元素。

p = uninitialized_fill_n(b,n,v)：用 v 初始化[b:b+n)中的每个元素。p=b+n。

p = uninitialized_copy(b,e,out)：用来自[b:e)中的相应元素初始化[out:out+(e−b))中的每个元素。p=b+n。

p = uninitialized_copy_n(b,n,out)：用[b:b+n)中的相应元素初始化[out:out+n)中的每个元素。p=b+n。

例如：

```
#include <vector>
#include <array>
#include <iostream>
#include <algorithm>
#include <functional>
#include <random>                                //伪随机数模块
using namespace std;
//生成随机整数的类
class Rand_int{
public:
    Rand_int(int lo, int hi) : p{ lo,hi } { }
    int operator()() const { return r(); }
private:
    uniform_int_distribution<>::param_type p;
    function<int()> r{ bind (uniform_int_distribution<>{p},
                            default_random_engine{}) };
};
//生成随机实数的类
class Rand_double{
public:
    Rand_double(double low, double high)
        :r(bind(uniform_real_distribution<>(low, high),
            default_random_engine())) { }
    double operator()() { return r(); }
private:
    function<double()> r;
};

int v1[3];
std::array<int,3> v2;
std::vector<double> v3;

template<typename C>
void Print(const C& c) {
for (auto &e : c)
    std::cout << e << '\t';
std::cout << '\n';
}
void main(){
    std::fill(std::begin(v1), std::end(v1), 9);  //v1 的所有元素值设置为 9
    //传递一个头等函数对象 Rand_int,v2 元素值设置为随机整数
    generate(begin(v2), end(v2), Rand_int(1,100));
    //输出 5 个[-100, 100]随机的浮点数
    generate_n(ostream_iterator<double>{cout,","}, 5, Rand_double(-100, 100));
    fill_n(back_inserter(v3), 5, 3.1);          //将 5 个 3.1 插入 v 的后面
    std::cout << std::endl;
    Print(v1);
    Print(v2);
```

```
    Print(v3);
}
```

执行程序,输出结果:

```
-72.9046,67.0017,93.7736,-55.7932,-38.3666,
9        9        9
13       3        35
3.1      3.1      3.1      3.1      3.1
```

程序中用 Rand_int()和 Rand_double()的函数对象分别生成一定范围的 int 或 double 类型数值。generate()每次调用这个函数对象时都会生成一个随机数值。uniform_int_distribution 和 uniform_real_distribution 是随机数模块< random >中的随机数发生器类模板,分别用来生成在一个数值范围内均匀分布的 int 和 double 随机类型数。

15. 合并:merge()

merge()将 2 个有序序列合并为一个有序序列。

```
merge (b1,e1,b2,e2,result);
merge (b1,e1,b2,e2,result,f);
```

将 2 个有序序列[b1,e1)、[b2,e2)中的元素合并到一个以 result 开头的有序序列中,f(e1,e2)是判断 e1 是否小于 e2 的比较谓词。

```
int first[] = {5,10,15,20,25};
int second[] = {50,40,30,20,10};
std::vector<int> v(10);
std::sort (first,first+5);
std::sort (second,second+5);
std::merge (first,first+5,second,second+5,v.begin());
```

下面是同样的在线版本的 merge()函数,可以作用于一个序列。

```
inplace_merge (b, m, e);
inplace_merge (b, m, e, f);
```

将[b:m)、[m:e)的元素合并到[b:e)中,f(e1,e2)是判断 e1 是否小于 e2 的比较谓词。

16. 堆操作:heap()

堆(heap)是一种特殊的序列,假如数据元素个数是 n 的序列 a,它的元素编号是从 1 开始的,即依次是 1、2、3、……。

(1) 如果任意编号 i 的元素满足:a[i]≤a[2i](若 2i≤n)且 a[i]≤a[2i+1](若 2i+1≤n),则这个序列称为小顶堆。例如:[5,8,22,9,23]就是一个小顶堆。

(2) 如果任意编号 i 的元素满足:a[i]≥a[2i](若 2i≤n)且 a[i]≥a[2i+1](若 2i+1≤n),则这个序列称为大顶堆。例如:[23,9,22,5,8]就是一个大顶堆。

小顶堆和大顶堆都是堆。对于小顶堆,第一个元素必然是所有元素中最小的,对于大顶

堆，第一个元素必然是所有元素中最大的。即小顶堆可以立即找到最小值，大顶堆可以立即找到最大值。而[5,23,22,9,8]既不是大顶堆也不是小顶堆。

```
make_heap(b, e);
make_heap(b, e, f);
```

这2个函数将一个序列[b,e)调整为一个堆。其中f是比较2个元素大小的谓词。

```
push_heap(b, e);
push_heap(b, e, f);
```

这2个函数用于将一个新元素插入序列的最后位置，即e－1指向的元素是插入的新元素，使[b:e)序列成为一个堆。

```
pop_heap(b, e);
pop_heap(b, e,f);
```

这2个函数删除堆序列的第一个元素，即将一个堆序列[b,e)中的第一个元素和最后一个元素交换，并使少了一个元素的序列[b,e－1)重新成为一个堆。

```
sort_heap(b, e);
sort_heap(b, e, f);
```

这2个函数对一个堆序列[b,e)进行排序，得到一个排序序列。

下面是一个综合性例子，说明这些函数的功能。

```
#include <iostream>
#include <algorithm>
#include <vector>
int main(){
    std::vector<int> v{ 5, 23, 22, 9, 8 };

    std::cout << "initially, v: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';

    std::make_heap(v.begin(), v.end());  //调整序列[v.begin(): v.end())为一个堆
    //std::make_heap(v.begin(), v.end(), [](double a, double b) {return b<a; });

    std::cout << "after make_heap, v: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';

    v.push_back(56);                     //将56追加到向量v后面

    std::cout << "before push_heap: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';
```

```
    std::push_heap(v.begin(), v.end()); //将最后一个元素即 v.end() - 1 的元素插入堆中

    std::cout << "after push_heap: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';

    std::pop_heap(v.begin(), v.end());  //将堆顶元素弹出,即第一个元素和最后一个元素交换
    auto largest = v.back();
    v.pop_back();                       //将 v 的最后一个元素删除
    std::cout << "largest element: " << largest << '\n';

    std::cout << "after removing the largest element, v: ";
    for (auto i : v) std::cout << i << ' ';
    std::cout << '\n';

    std::sort_heap(v.begin(), v.end());

    std::cout << "sorted:\t";
    for (const auto &i : v) {
        std::cout << i << ' ';
    }
    std::cout << '\n';
}
```

17. 集合操作

集合操作指集合的并、交、差等运算以及判断一个集合是否是另一个集合的子集。这里的集合用一个序列[b,e)表示。用于并、交、差、对称差运算的函数分别是 set_union()、set_intersection()、set_difference()、set_symmetric_difference(),而函数 includes()用于判断一个集合是否是另一个集合的子集。

```
includes (b1, e1, b2, e2);
includes (b1, e1, b2, e2, comp);
```

如果一个序列[b1,e1)中包含了序列[b2,e2)中的元素,includes()函数返回 true,否则返回 false。2 个元素 a、b 相等的条件是:if! (a<b) && ! (b<a))或者 if (!comp(a,b) && !comp(b,a))。

```
set_union (b1, e1, b2, e2, result);
set_union (b1, e1, b2, e2, result, comp);
```

set_union()将两个有序序列[b1,e1)、[b2,e2)合并为一个新的序列 result,如果一个元素在 2 个序列都出现,则该元素出现次数最多的这个元素子序列将出现在最终的结果序列中。如:

```
std::vector<int> v1 = { 1, 2, 3, 4, 5, 5, 5 };
std::vector<int> v2 = { 3, 4, 5, 6, 7 };
```

```
std::vector<int> dest1;
std::set_union(v1.begin(), v1.end(),v2.begin(), v2.end(),  std::back_inserter(dest1));
for (const auto &i : dest1) {   std::cout << i << ' ';   }
std::cout << '\n';
```

输出的结果是：

```
1 2 3 4 5 5 5 6 7
set_intersection (b1, e1, b2, e2, result);
set_intersection (b1, e1, b2, e2, result, comp);
```

set_ intersection ()求两个有序序列[b1,e1)、[b2,e2)的交集，结果为序列 result。

```
set_difference (b1, e1, b2, e2, result);
set_difference (b1, end1, b2, e2, result, comp);
```

set_ difference ()求两个有序序列[b1,e1)、[b2,e2)的差集，结果为序列 result。

```
set_symmetric_difference (b1, end1, b2, e2, result);
set_symmetric_difference (b1, end1, b2, e2, result, comp);
```

set_symmetric_difference ()求两个有序序列[b1,e1)、[b2,e2)的对称差集，结果为序列 result。

13.5 智能指针

13.5.1 raw 指针和智能指针

前面用 new 运算符分配一块内存，可以将这块内存的地址保存在一个指针变量中：

```
auto *p{ new int }, *q{ new int[3] };     //p、q 指向动态分配内存
```

指针变量不但可以指向动态分配的内存，也可以指向一个自动变量：

```
auto i{ 3 };
p = &i;                                   //p 也可以指向一个 int 类型的变量
```

即一个 T * 类型的指针变量可以存储动态分配的内存地址，也可以存储一个程序块的普通变量的地址。这种指针称为 **raw 指针**(原始指针)，因为除了内存地址外它没有存储别的东西。

raw 指针指向的动态分配内存应该及时释放，并且对 new 分配的内存要用 delete 释放，对于 new[]分配的内存要用 delete[]释放。如果一个指针变量在指向新的内存时，没有释放原先指向的动态内存，就会造成**内存泄漏**(memory leaks)。例如，上述代码中 p 开始指向的是 new 分配的动态内存，然后修改 p 指向了变量 i，而原先 p 指向的内存并没有用 delete

释放，这块动态内存就一直被程序占用，程序的其他部分或其他程序没法访问或释放这块内存，造成了内存泄漏。

同样，如果用错了 delete 也会造成内存汇漏，例如：

```
delete q;              //错：只释放了 q 指向的 3 个 int 内存块中的第 1 个 int 内存块
q = 0;
```

q 本来存储的是用 new[]分配的可存储 3 个 int 类型的内存块地址，但 delete q 只释放了第 1 个 int 内存块，导致另外 2 个 int 内存块无法释放，也造成了内存泄漏。

在程序中，指针变量通过赋值语句、将函数参数不断赋值给其他变量，这些指针变量可能散落在程序的不同函数中，如何保证正确、及时地释放一块动态内存？需要程序员非常小心。即使这样也不可避免会导致内存泄漏或一块动态内存被多次释放，或释放的不是一块动态内存等很多问题。例如：

```
delete p;              //错：p 指向的不是动态内存
```

因为之前 p 指向了变量 i，而 i 占用的不是动态内存，delete p 也会导致程序崩溃。同样，如果多次释放同一块动态内存也会引起同样的问题而导致程序崩溃。

另外，假设有 2 个指针指向同一块内存，如果通过一个指针释放了这块内存空间，但又通过另外一个指针访问这块内存空间，也会引起非法内存访问的严重问题。

```
p = new int;
q = p;
delete p; p = 0;
auto j{ *q };
```

其中，p、q 指向的是同一个内存块，然后通过 delete p 释放了这块内存，但 q 不知道这块内存已经被释放，继续用 *q 去获取这块内存中的值就导致非法内存访问。

直接使用 raw 指针经常会导致内存泄漏、非法内存访问、多次释放同一块动态内存、释放非动态内存等问题，即使很有经验的程序员也不可避免会犯上述错误。对于不熟练的程序员，raw 指针很难使用。

为解决直接使用 raw 指针的困难和引起的问题，C++通过类模板提供了更加方便的对 raw 指针包裹的智能指针。作为类对象，智能指针不仅包含内存地址，还有一些其他信息或功能，可以避免直接使用 raw 指针带来的问题和困难，例如使用智能指针不需要程序员显式调用 delete 运算符释放指针指向的动态内存，智能指针会自动释放不再使用的内存。智能指针在使用上类似于 raw 指针，也很容易使用。

智能指针有 3 种：shared_ptr、unique_ptr 和 weak_ptr。它们定义在头文件< memory >中，因此，使用它们必须包含头文件< memory >。当然它们的名字也都在标准名字空间 std 中。

13.5.2 unique_ptr

std::unique_ptr<>对象是对 raw 指针的包裹，将一个 T * 类型的动态分配内存块的 raw

指针作为 std::unique_ptr<T>构造函数的参数就可以创建一个 std::unique_ptr<T>对象，在它退出其作用域销毁时，析构函数会自动释放其包含的 raw 指针指向的动态内存。

```
#include <memory>
#include <iostream>
void f() {
    std::unique_ptr<double> p{ new double{0.} };
    *p = 3.14;
    std::cout << *p << '\t';
}
int main() {
    f();
}
```

上述程序用 std::unique_ptr<double>类对象 p 包裹了动态分配的内存{new double{0.}}。当 f()函数结束时，p 被销毁，其析构函数会自动释放动态分配的内存。p 的使用和 raw 指针一样，即可以用解引用运算符 *（或间接访问运算符->）访问 p 指向的动态内存。

执行程序，输出结果：

```
3.14
```

作为一个类对象，p 可以使用 unique_ptr<>类的成员函数。如可以用 get()方法得到其包裹的 raw 指针：

```
double *rp = p.get();
*rp = 3.1415;
std::cout << *(p.get()) << '\t';
```

执行程序，输出结果：

```
3.1415
```

还可以用 reset()将一个新的 raw 指针传递给它，原来的 raw 指针指向的内存被自动释放，例如：

```
std::cout << p.get() << '\t';
p.reset(new double);
std::cout << p.get() << '\n';
```

执行程序，输出结果：

```
009CD180        009CCE38
```

可以看到，执行 p.reset(new double)后的 raw 指针指向了一块新的内存，而原先 raw 指针指向的内存被释放了，不会造成任何内存泄漏。

如果不传递给 reset()方法任何参数，则释放 raw 指针占用的内存后，将 raw 指针设置

为空指针 nullptr。例如：

```
p.reset();
std::cout << p.get() << '\n';
```

执行程序,输出结果：

```
00000000
```

也可以用 release()方法返回 raw 指针并将 raw 指针设置为空指针 nullptr。例如：

```
p.reset(new double);
double * rawp = p.release();
*rawp = 3.0;
std::cout << p.get() << '\n';
```

其中,p.release()将 raw 指针作为返回值赋值给 rawp,并将 p 内部的 raw 指针设置为空指针 nullptr。因此,执行程序,输出结果：

```
00000000
```

当然可以用 std::unique_ptr<T[]>指向一个 new T[]分配的动态数组空间：

```
std::unique_ptr<char[]> p(new char[5]);
p[0] = 'A'; p[1] = 'B'; p[2] = 'C';
for (auto i = 0; i != 3; i++)
    std::cout << p[i];     //和 raw 指针一样,可以用过下标访问 p 指向的动态数组的元素
std::cout << std::endl;
```

执行程序,输出结果：

```
ABC
```

1. std::make_unique<>

除直接将一个 raw 指针传递给 std::unique_ptr<>的构造函数外,C++还提供了 std::**make_unique**<>()函数模板可以很方便地帮助创建一个 std::unique_ptr<>指针并分配动态内存：

```
auto q = std::make_unique<double>(3.14);  //分配 1 个 double 动态内存块的 unique_ptr 指针
std::cout << *q << '\n';
```

当然,可以分配一个动态数组空间：

```
auto p = std::make_unique<double[]>(3);    //分配 3 个 double 动态内存块的 unique_ptr 指针
p[0] = 'A'; p[1] = 'B'; p[2] = 'C';
for (auto i = 0; i != 3; i++)
```

```
        std::cout << p[i];
    std::cout << std::endl;
```

std::unique_ptr 是一个**独享指针**,即不能将它复制给其他变量,也即不能使用拷贝构造函数和复制运算符:

```
auto p = std::make_unique<double>(3.14);
auto q{ p };                              //错: 不能拷贝构造
std::unique_ptr<double> r{};
r = p;                                    //错: 不能赋值
```

但 std::unique_ptr<>支持移动语义。

2. std::unique_ptr<>

那么如何将一个 std::unique_ptr<>传递给函数或作为函数返回值返回呢?因为 std::unique_ptr<>支持移动语义,因此,可以直接返回这个 std::unique_ptr<>对象:

```
std::unique_ptr<int> get_unique() {
    auto ptr = std::unique_ptr<int>{ new int{2} };
    return ptr;                           //ptr 被 move(移动)到临时的返回结果中
}
void f() {
    //...
    auto uptr = get_unique();             //get_unique()的返回值被 move(移动)到 uptr 中
    //...
}
```

std::unique_ptr<>对象可以作为函数的实参传递给函数,必须先将左值转换为右值,然后执行同样的移动语义。

```
void fun(std::unique_ptr<int> ptr){
    //...
}

int main(){
    std::unique_ptr<int> p = get_unique();
    fun(p);                               //错: 左值不能隐含地调用 move()构造函数
    fun(std::move(p));                    //可以。std::move()将左值转换为右值引用
    return 0;
}
```

上述代码中 fun(std::move(p))将 p 转换为右值引用,然后通过移动语义将其 raw 指针移交给 fun()的参数 ptr。更好的方法是将 fun()的形参定义为引用形参,就可避免移动操作。

```
void fun(std::unique_ptr<int> &ptr){
    //...
}
```

13.5.3 shared_ptr

和 std::unique_ptr<>一样，std::shared_ptr<>也是对 raw 指针的包裹，并可以类似地使用。但 std::shared_ptr<>指针可以通过拷贝构造函数或赋值运算符任意地复制，这些复制的对象将共享同一个 raw 指针，即它们指向同一块动态内存，因此，可以通过任何一个 std::shared_ptr<>指针访问这块内存。所以，std::shared_ptr<>指针被称为**共享指针**。因为多个 std::shared_ptr<>指针共享同一个 raw 指针，所以 std::shared_ptr<>内部维护了一个引用计数器，表示共享这个内存块的 std::shared_ptr<>指针的个数，当一个 std::shared_ptr<>指针内部的这个引用计数器变为 0 时，std::shared_ptr<>的析构函数才真正释放这块内存，如果没有变为 0，析构函数做的工作仅仅是将计数器减少 1。

```
#include <iostream>
#include <memory>
#include <string>
int main() {
    auto ptr = std::make_shared<std::string>("hello");  //ptr 是指向 string 动态内存块
                                                         //的共享指针
    *ptr = "world";                                      //ptr 指向的 string 内存块内容
                                                         //修改为"world"
    std::cout << *ptr << '\t'<< ptr.use_count()<< '\n';  //ptr 指向的内存块引用计数为 1
    auto ptr2 = ptr;                                     //用 ptr 拷贝构造 ptr2,它们指
                                                         //向的同一个动态内存块的引用
                                                         //计数变为 2
    *ptr2 = "hello world";
    std::cout << *ptr << '\t'<< ptr.use_count() << '\n';
    std::cout << *ptr2 << '\t'<< ptr2.use_count() << '\n';
    ptr.reset();                                         //ptr 设置为一个空指针,原先内
                                                         //存块的引用计数减少 1
    std::cout << ptr << '\n';
    std::cout << *ptr2 << '\t'<< ptr2.use_count() << '\n';
}
```

程序中通过 unique_ptr<>的成员函数 use_count()可以查询有多少 unique_ptr<>共享动态内存块。reset 释放原来的 raw 指针只是减少 ptr 和 ptr2 共享内存的引用计数器，并没有真正释放内存。

执行程序，输出结果：

```
world   1
hello world    2
hello world    2
0
hello world    1
```

同样，可以用 reset()成员函数释放原来的 raw 指针，让 raw 指针指向新的动态分配的

对象。

```
ptr2.reset(new std::string{"wang"});     //释放原来的 raw 指针,指向新 string 对象
std::cout << *ptr2 << '\t'<< ptr2.use_count() << '\n';
```

执行程序,输出结果:

```
wang   1
```

std::shared_ptr<>的引用计数是原子操作,因此是线程安全的。对于共享同一个对象的多个线程,可以为每个线程定义一个 std::shared_ptr<>,这些 std::shared_ptr<>共享同一个对象就可以使每个线程都能访问这个对象。

13.5.4 weak_ptr

std::weak_ptr<>称为**弱指针**,是一个配合 std::shared_ptr<>指针的智能指针。std::shared_ptr<>是一种具有所有权的指针,每个 std::shared_ptr<>都拥有它指向的对象的所有权,这个所有权是通过引用计数实现的,即每创建一个新的 std::shared_ptr<>,这个新的 std::shared_ptr<>就使它拥有的对象的引用计数增加 1,而销毁一个 std::shared_ptr<>就使它拥有对象的引用计数减少 1,当引用计数变为 0 时,即没有 std::shared_ptr<>拥有这个对象时,该对象才被真正释放。

std::weak_ptr<>只能从一个 std::shared_ptr<>创建,它是对由 std::shared_ptr<>管理的对象的非拥有(弱)引用,即 std::weak_ptr<>不拥有 std :: shared_ptr<>管理的对象。

用 std::weak_ptr<>可以查询 std::shared_ptr<>管理对象,即 std::weak_ptr<>是 std::shared_ptr<>的一个观察者。std::weak_ptr<>观察的对象可能已经被 std::shared_ptr<>销毁,如果没有被销毁,则 std::weak_ptr<>可以转换为 std::shared_ptr<>而承担起临时所有权,如果之后销毁了原始的 std::shared_ptr<>,则会延长对象的生命周期,直到临时的 std::shared_pt<>被销毁为止。

std::weak_ptr<>的 lock()可以得到其观察的 std::shared_ptr<>对象的临时所有权。

```
#include <iostream>
#include <memory>
std::weak_ptr<int> gw;
void observe(){
    std::cout << "use_count == " << gw.use_count() << ": ";
    if (auto spt = gw.lock()) {       //gw.lock()的结果必须复制到一个 shared_ptr 才能使用
        std::cout << *spt << "\n";
    }
    else {
        std::cout << "gw is expired\n";
    }
}
```

```
int main(){
    {
        auto sp = std::make_shared<int>(42);
        gw = sp;
        observe();
    }
    observe();
}
```

上述代码中,gw = sp 使得 gw 成为 sp 的一个观察者,在第 1 次调用 observe()时,gw.lock()得到了临时所有权并复制到一个 shared_ptr<>变量 spt 中,然后就可以通过 *spt 访问 spt 指向的对象了。在该函数结束后,这个临时的 spt 被销毁。

当 sp 退出其作用域时,其指向的对象的引用计数变为 0,其指向的对象就被完全销毁了。第 2 次调用 observe()时,gw 绑定的 sp 拥有的对象已经被销毁,因此引用计数为 0。这样也就无法 lock()到一个 shared_ptr。

执行程序,输出结果:

```
use_count == 1: 42
use_count == 0: gw is expired
```

13.6 字符串

13.6.1 字符:<cctype>、<cwctype>

<cctype>(移植自 C 语言的 ctype.h)和<cwctype>(移植自 C 语言的 wctype.h)声明了一组用于对单个 ASCII 字符和 wchar 宽字符进行分类和转换的函数。如:

int isalpha (int c):检查字符是否是字母。

int iswalpha (wint_t c):检查宽字符是否是字母。

int ispunct (int c):检查字符是否是标点字符。

int iswpunct (wint_t c):检查宽字符是否是标点字符。

int toupper (int c):将小写字符转换为大写。

wint_t towupper (wint_t c):将小写宽字符转换为大写。

完整的函数列表请参考:http://www.cplusplus.com/reference/cctype/和 http://www.cplusplus.com/reference/cwctype/。

13.6.2 C 风格字符串

C 风格字符串是以空字符'\0'(ASCII 值是 0)结尾的字符数组。头文件 cstring(移植自 C 语言的 string.h)定义了处理 C 风格字符串的函数。表 13-2 所示是其中的几个常用函数。

表 13-2 C 风格字符串常用函数

函数规范	说明	例子
size_t strlen(const char * s)	返回 s 的长度,不包括终止空字符'\0'。size_t 通常是 unsigned int 的 typedef	char * s= "Hello"; cout << strlen(s);
char * strcpy (char * dest, const char * src)	将 src 复制到 dest,返回 dest	char s1[]="hello",s2[10]; strcpy (s2,s1);
char * strncpy (char * dest, const char * src,size_t n)	将最多 n 个字符从 src 复制到 dest,返回 dest	
char * strchr (char * s,int c);	返回指向 s 中第一个出现字符 c 的指针	
char * strstr (char * s1, char * s2);	返回指向 s1 中第一次出现 s2 的指针	char s[] ="This is a cat"; char * p= strstr (s,"cat"); cout << p-s;

此外,在<cstdlib>还包含了将 C 风格字符串转换为基本类型的函数,如表 13-3 所示。

表 13-3 C 风格字符串转换为基本类型的函数

函数规范	说明	例子
int atoi (char * s)	将 s 解析为 int	int i = atoi ("25");
double atof (char * s)	将 s 解析为 double	

更完整的函数列表及说明请参考网上相关文档。

13.6.3 C++的字符串

C++提供了对各种类型字符处理的功能强大的字符串类模板 basic_string:

```
template < class charT,
    class traits = char_traits < charT >,     //basic_string::traits_type
    class Alloc = allocator < charT > >       //basic_string::allocator_type
class basic_string;
```

basic_string 模板类有 4 个实例:

```
using std::string = std::basic_string < char >;
using std::wstring = std::basic_string < wchar_t >;
using std::u16string = std::basic_string < char16_t >;     //C++11 之后
using std::u32string = std::basic_string < char32_t >;     //C++11 之后
```

basic_string 实现了很多成员函数模板以及重载了运算符对字符串对象进行操作。下面以 char 类型的 string 类为例介绍这些函数模板和重载运算符。

1. 构造函数

多个不同的构造函数可用于创建一个字符串。

```
#include <iostream>
#include <string>                          //C++ string 类头文件
using namespace std;
int main() {
    string s1, s2("hello"), s3 = "world";
    string s4(s3);                         //拷贝构造函数
    string s5 = {'c','+','+'},s6{"python"};  //初始化列表
    string s7(8, 'a');                     //8 个字符'a'构成的字符串
}
```

2. 重载的运算符

很多运算符都被重载以处理字符串对象、C 风格字符串和文字量。

(1) 作为成员函数重载的运算符。例如：

```
//第一个操作数必须是 string 类对象
=                                //赋值
[]                               //下标访问(不检查下标是否合法)
+=                               //在原字符串后面加上另一个字符串
```

(2) 作为(非成员函数)友元重载的运算符。第一个操作数不必是 string 类对象。

```
+                        //拼接 2 个字符串(其中一个可以是 C 风格串或文字量)
                         //返回拼接后的新的 string 对象
==, !=, <, <=, >, >=     //关系(比较)运算符
                         //其中一个操作数可以是 C 风格串或文字量
>>                       //流输入运算符
<<                       //流输出运算符
```

3. 公开的成员函数

公开的成员函数很多，限于篇幅，这里只列举一些常见的函数，如表 13-4 所示。

表 13-4 string 类的一些成员函数

函数规范	说明
size()	返回字符串中字符个数
capacity()	存储空间的大小(容量)
resize()	改变字符串的大小
reserve()	改变存储空间容量
clear()	清空字符串，但不改变容量
empty()	判断是否是空串(大小为 0)
at()	下标访问(检查下标是否合法)
front()	第一个字符
back()	最后一个字符
push_back()	添加字符到尾部
pop_back()	删除最后一个字符
insert()	在某位置插入一个字符或字符串

续表

函 数 规 范	说　　明
erase()	删除某位置或范围的字符
substr()	求子串
replace()	替换子串
swap()	交换字符串的值
c_str()	返回 C 风格字符串
copy()	从字符串中复制一个子序列
find/rfind()	正向或逆向查找一个字符串
find_first_of()	查找第一次匹配字符的位置
find_first_not_of()	查找第一次未匹配字符的位置
find_last_of()	从尾部查找第一次匹配字符的位置
find_last_not_of()	从尾部查找第一次未匹配字符的位置
begin()/cbegin()	返回第一个元素位置的迭代器
end()/cend()	返回最后元素的后一个位置的迭代器
rbegin()/crbegin()	逆向的第一个元素迭代器
rend()/crend()	逆向的最后元素的后一个位置迭代器

如 string 类的 begin()、end()成员函数可返回首字符和尾字符的后一个位置的迭代器：

```
#include <iostream>
#include <string>
#include <cctype>
int main(){
    std::string str("hello  world");
    for (std::string::iterator it = str.begin(); it != str.end(); ++it)
        *it = toupper(*it);
    for (std::string::iterator it = str.begin(); it != str.end(); ++it)
        std::cout << *it;
    std::cout << '\n';
    return 0;
}
```

执行程序，输出结果：

```
HELLO  WORLD
```

另外，string 类还有一个公开的静态变量 string::npos 表示字符串长度的可能最大值，通常就是 size_t 类型的最大值。find_first_of()如果返回这个值，表示未找到一个子串。例如：

```
#include <iostream>     //std::cout
#include <string>       //std::string
#include <cstddef>      //std::size_t

int main(){
```

```
    std::string str("it is a cat,that is a dog ");
    std::size_t found = str.find_first_of("is");
    while (found != std::string::npos)      {          //如果不相等,则表示找到
        str[found] = '*';
        found = str.find_first_of("is", found + 1);    //寻找下一个"is"中的字符
    }
    std::cout << str << '\n';
}
```

该程序将 str 中查找到的字符'i'或's'都替换为'*'。执行程序,输出结果:

```
*t ** a cat,that ** a dog
```

可以看到,和"is"中字符匹配的字符都被替换成了'*'。

除通过>>和<<从流中输入或向流出字符串,string 的友元函数 getline()可以从流中读取一行字符串。例如:

```
#include <iostream>
#include <string>
int main(){
    std::string name;
    std::cout << "请输入你的名字: ";
    std::getline(std::cin, name);
    std::cout << "Hello, " << name << "!\n";
    return 0;
}
```

执行程序,输出结果:

```
请输入你的名字: Li Ping
Hello, Li Ping !
```

4. 字符串视图

此外,C++17 的类模板 std::basic_string_view(字符串视图)提供了一个轻量级对象,它使用类似于 std :: basic_string 接口的接口提供对字符串或其子串的只读访问。

```
using std::string_view = std::basic_string_view<char>;
using std::wstring_view = std::basic_string_view<wchar_t>;
using std::u16string_view = std::basic_string_view<char16_t>;
using std::u32string_view = std::basic_string_view<char32_t>;
```

basic_string_view 是观察 basic_string 字符串的一个窗口,其目的是对字符串操作时避免不必要的字符串复制,如字符串的 substr()成员函数获得的子串没有单独分配内存,只是指向原来字符串的子串部分。

下面的程序通过重载 new 内存分配运算符,查看 basic_string 和 basic_string_view 的 substr()成员函数是否申请了动态内存。

```
#include <string>
#include <iostream>
#include <string_view>

//需要重载 new 运算符来查看到底有没有分配内存
void* operator new(std::size_t n){
    std::cout << "new " << n << " bytes\n";
    return malloc(n);
}

int main(){
    std::string str{ "This is a cat" };     //原始的字符串,分配一次内存

    //求子串时,又会分配内存,并执行复制操作
    auto subStr = str.substr(str.find("cat"));
    std::cout << subStr << "\n";
    std::cout <<"-----------\n";
    //求子串,没有内存分配
    std::string_view strView{ str };
    auto subView = strView.substr(str.find("cat"));
    std::cout << subView << "\n";
}
```

执行程序,输出结果:

```
new 8 bytes
new 8 bytes
cat
-----------
cat
```

可以看到,使用 string_view 可避免内存分配等耗时操作,提高了程序效率。string_view 不但可以观察 string,还可以接受原始的 C 风格字符串,如 char const *、std::vector<char>向量等,并避免内存分配和复制的开销。函数的形参建议尽量用 string_view 代替 const string&,可以避免从 C 风格字符串隐式类型转换构造一个 string 对象的开销。下面的代码在将一个 C 字符串传递给函数 f()时,会调用 std::string 的构造函数,即需要申请一块动态内存。

```
void f(const std::string& s){
    /* … */
}
int main(){
    f("hello, world!");     //创建一个 std::string 对象,需要动态内存分配
    char msg[] = "good morning!";
    foo(msg);               //创建一个 std::string 对象,需要动态内存分配
}
```

如果将 f 的形参换成 std::string_view 就可以避免动态内存分配。

13.7 习题

1. 输入流的 get(char * s,streamsize n,char delim)遇到分隔符 delim 时就停止,分隔符仍然保留在输入流中。假如执行下列程序,输入的内容是"hello# world#",程序的输出是什么? 如何在下次读取时跳过分隔符?

```
#include <iostream>
#include <fstream>
int main() {
    char str[256],str2[256];
    std::cout << "输入包含 2 个#的字符串: ";
    std::cin.get(str, 256,'#');
    std::cin.get(str2, 256, '#');
    std::cout << str <<'\n';
    std::cout << str2 << '\n';
}
```

2. 下面的 copyFile()是文件复制函数,它有什么缺点? 如何改进?

提示:

(1) 可以用 get()或 putc()。

(2) 使用二进制读写函数 read()和 write()。

```
void copyFile (const std::string filename1, const std::string filename2) {
    std::ifstream file1(filename1);
    std::ofstream file2(filename2);
    std::string line;
    if (file1.good() && file2.good()) {
        while (getline(file1, line)) {
            file2 << line;    file2 << '\n';
        }
    }
    file1.close();    file2.close();
}
```

3. 假如有一组 Date 类(见第 7 章)对象,要求:

(1) 从键盘输入一组 Date 信息,将它们保存到一个 list 对象中。

(2) 将该 list 中的 Date 对象按照每个 Date 对象一个数据块的形式写入一个二进制文件中。

(3) 将保存在文件的第 2、5、7 个 Date 对象读入到一个 vector 对象中,并显示出来。

(4) 将第 5 个 Date 对象保存回原来文件中的第 5 个 Date 对象的位置。

(5) 再从文件中将第 5 个 Date 对象读出并显示。

对于一般的类(如表示学生成绩信息的 Student 类),上述操作的代码是否适用? 为

什么？

4. 编写一个通讯录程序，从键盘输入一个人的姓名和他的0个或多个电话号码，将它们写入一个文本文件中，每个人的信息单独放在一行。然后输入一个查找的姓名，从文件中依次读取信息并和待查找的姓名比较，如果找到，则显示这个人的姓名和其所有电话号码，如果没找到，给出“未找到”的提示。

5. 编写一个程序，用getline()从第4题的文件中读入每行字符串，用这个字符串构造一个istringstream输入字符串流对象，然后用>>抽取其中的姓名和电话号码，并验证电话号码是否是合法的电话号码(如11～12位固话或11位手机号码)，舍弃非法电话号码，然后格式化该人姓名和合法电话号码并放入一个ostringstream对象中，使得姓名后添加一个冒号，电话号码之间添加一个逗号。最后将格式化的通讯录输出到另外一个文件中。

6. 对于下面的任务，vector、deque、list哪种最适合？为什么？

(1) 读取一组单词，将它们按照字典顺序插入在容器中。

(2) 读取未知数量的单词，总是将新单词放在最后，删除操作总是删除最前面的单词。

(3) 从键盘或文件读取未知数量的实数，将它们排序并输出。

7. 下列代码中有没有错误？为什么？如有错误请纠正它。

```
std::list<int> lst1;
std::list<int>::iterator iter1 = lst1.begin(), iter2 = lst1.end();
while (iter1 < iter2) { /* ... */}
```

8. 下面4个对象的类型是什么？

```
std::vector<int> v1;
const std::vector<int> v2;
auto it1 = v1.begin();
auto it2 = v2.begin();
auto it3 = v1.cbegin();
auto it4 = v2.cbegin();
```

9. 可以直接用==运算符比较同类型容器的对象是否相等(即内容是否相同)，但对不同类型的容器则不能这样做，请编写一个函数，比较一个vector和一个list对象的对应元素是否相等，并测试该函数是否正确。

10. 编写一个程序，从键盘输入一系列整数，将它们依次放入一个队列中，然后再将该队列中的元素依次取出(即删除)，将这些取出的整数分别放入另一个队列中，使得所有奇数都在偶数的左边。

11. 将数组[34,2,16,28,19,11]中的所有整数用vector和list的assign成员函数分别复制到vector和list对象。然后分别删除vector对象中的所有奇数和list对象中的所有偶数。在删除操作之前和之后，输出这2个容器对象中的所有元素。

注： 关于assign成员函数，请网上搜索其含义和用法。

12. std::vector的reserve()和resize()成员函数的含义和区别是什么？

13. vector的capacity()方法的含义是什么？为什么list和array没有capacity()方法？

14. 从键盘输入一组单词，在输入每个单词(每个单词是一个string对象)时，用插入排

序的思想将该单词插入一个 list 对象中，以便这些按照字典顺序存储在这个 list 对象中，最后输出 list 中所有的字符串，查看是否是按字典顺序排序的。

15. 编写一个普通函数或函数对象作为 std::min_element 模板的比较谓词，返回离输入值最远的元素。

16. 用 accumulate()算法计算一个 list<double>中的所有实数之和。

17. 使用 find_if()在一组学生的 vector 中(如 vector<Student>)查找某个名字的学生信息。

18. 编写一个程序，统计从键盘输入的每个单词出现的次数。

19. 从键盘输入一组单词，将它们放入一个 vector 对象中，遍历这个 vector，将所有单词的首字母改成大写，最后分别按照字典顺序和字符串长度排序并输出排序的结果。

20. 实现一个模拟 C 风格字符串拼接函数 strcat()的函数 Strcat()。

21. 编写一个程序，用 getline()读入一个字符串，将其中的标点符号都替换成'#'，然后输出这个修改后的字符串。

22. 编写一个程序，从键盘输入一系列单词，直到结束符(Ctrl+Z(Windows)或 Ctrl+D(UNIX))结束输入，然后按照长度统计每个长度的单词数目。

提示：用关联数组 map。

23. 一个 list 中包含整数 1～9，使用 inserter()、back_inserter()和 front_inserter()将它们插入其他 3 种不同容器中。查看结果是否符合你的预期。

24. 使用绑定输入流对象 std::cin 的流迭代器读取来自标准输入的一系列整数，并用 sort()对这些整数排序，然后复制到绑定输出流对象 std::cout 的流迭代器上，即将排序的整数序列写回标准输出。如果希望输出整数序列中不能有重复的整数，又该怎么办？

提示：输入的整数可以保存在一个 vector<int>对象中。

25. 编写程序，输入 3 个文件名：1 个输入文件名、2 个输出文件名。输入文件包含字符串和实数。使用 istream_iterator 读取输入文件，使用 ostream_iterators 将字符串和实数分别写到 2 个不同的输出文件中。

26. 将 13.3.2 节的 X 类换成一个占用资源的类，如 string 或 vector，并通过执行时间等方法说明移动迭代器的作用。

27. 编写代码用 for_each()算法计算一组实数的平均值。

28. 下面代码的错误原因是什么？

```
int first[] = {5,10,15,20,25};
int second[] = {50,40,30,20,10};
std::vector<int> v(10);
std::merge (first,first+5,second,second+5,v.begin());
```

29. 下列关于 std::unique_ptr 的哪些语句是错误的？为什么？

```
std::unique_ptr<int> p1(new int());
std::unique_ptr<int> p2 = new int();
std::unique_ptr<int> p3(p1);
std::unique_ptr<int> p4 = p1;
```

```
int i{2};
std::unique_ptr<int> p(&i);
```

30. unique_ptr 可以通过 release()将所有权转移给另外一个 unique_ptr,请举例说明如何使用 release()。为什么 share_ptr 没有这个函数?

31. 下列程序有什么问题?

```
auto sp = make_shared<int>();
auto p = sp.get();
delete p;
```

32. 以下哪个 unique_ptr 声明是非法的或可能导致后续程序错误?为什么?

```
int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef unique_ptr<int> IntP;
```

(1) IntP p0(ix);

(2) IntP p1(pi);

(3) IntP p2(pi2);

(4) IntP p3(&ix);

(5) IntP p4(new int(2048));

(6) IntP p5(p2.get())。

第14章

异 常 处 理

14.1 错误和异常处理

14.1.1 错误的分类

C++程序的错误可分为 2 类，其中一类是**语法错误**（也称**编译时错误**），即编译程序时编译器发现的违背编程语言语法规则的错误，初学者因为不熟悉 C++语法，编写的程序中会出现很多的语法错误，只要根据编译器提示的错误原因，一般很容易在错误提示的行或前面的行找出错误的原因，即语法错误很容易纠正。随着对语言越来越熟悉，编写程序的语法错误会越来越少。另一类是**运行时错误**，即程序可以编译运行，但在运行过程中会出现意想不到的结果甚至崩溃。运行时错误包括异常和逻辑错误，逻辑错误是指程序设计的逻辑存在某种问题，如求 2 个数的平方和被写成了求 2 个数的和的平方，这种错误不会引起任何异常的警告或报错，程序似乎一切运行良好，但是结果不对。当然有时逻辑错误也会导致异常警告或报错。这种程序运行过程中导致异常情况甚至程序崩溃的错误称为**异常**。如程序中一个数除以 0、读写一个不存在的文件、读写网络时网络突然断开等都会引起异常错误。

14.1.2 传统的错误处理方法

传统的错误处理分为 3 种。

（1）一种处理方法是忽视它，程序继续执行，但通常这会继续导致后续代码出现更多的异常，如对一组数组中的数求平均，如果发现数组没有任何数据元素，还继续去求平均，会进一步导致其他异常。

（2）处理程序异常的最简单粗暴的方法就是直接中止程序。除特殊情况外，一般情况下应该避免这样做。

（3）最常使用的错误处理方法是设置一个错误码，并中断当前函数的执行，回退到当前函数的调用函数中。如果当前函数是 main()主函数，则中断程序执行并返回错误码给操作

系统。如果发生错误的函数不是主函数,则从当前函数回退到它的调用函数,调用函数可根据当前的错误码做适当的处理。

错误码通常是一个整数,每个整数表示一个具体类型的错误。发生错误的函数有 2 种返回错误码的方式,其中一种是设置一个全局的错误码。因为全局错误码只能表示一个错误,如果有多个错误发生,后出现的错误就会覆盖掉之前的错误码,而之前的错误可能仍然存在。

不同于全局错误码,一般的函数可通过返回一个错误码给调用者,表示函数执行过程中发生了某种错误。这就需要每个调用函数通过检查被调用函数返回的错误码来判断是否发生了某种错误。如:

```
if( fun()!= 0){      //假设 fun()函数返回 0 表示没有任何错误
      //错误处理代码
}
```

每次函数调用都进行这种错误检查会使程序代码不断增多,降低了程序的可读性。实际编程中程序员经常忽略检查函数是否成功,但这样一来就会遗漏未被处理的错误。

返回错误码还有一个问题,因为错误码通常是一个整数,假如一个函数本身也需要返回一个整数,函数返回值既要表示函数执行结果又要表示是否发生错误,就会产生冲突。一种解决方法是通过正负整数来区分,如小于 0 的整数就表示错误码,大于 0 的整数就表示返回值,但如果函数返回值也是负整数呢?一种解决方法是将错误码或返回值通过函数的参数(如引用参数或指针参数)返回给调用者。

14.1.3 C++的异常处理

C++提供的异常处理机制将正常代码和异常处理代码分开,使得程序代码更简单、清晰并且不会遗漏错误(异常)。

C++异常处理的基本思想是一个函数发现了一个自己无法处理的异常情况,它会抛出(用关键字 **throw**)一个异常对象(异常对象可以是任何类型的变量(对象)),该函数希望它的调用者(上级)能处理这个异常。由于异常是一个对象,里面可以包含很多关于异常的信息而不仅仅是一个难以理解的错误码。

下面代码中的 do_task()是执行某个任务的函数,如果处理任务正常,就返回一个结果 result;如果发生了某种异常(错误),就抛出(throw)一个叫作 Some_error 的异常(错误)对象。

```
int do_task(){
    //...
    if (正常处理了某些工作)
        return result;       //返回结果
    else
        throw Some_error{}; //抛出错误
}

void taskmaster(){
```

```
        try {
            auto result = do_task();
            //使用返回的结果 result
            //… 其他处理
        }
        catch (Some_error) {
            //do_task 发生了某种异常：处理这个异常
        }
    }
```

taskmaster()函数执行过程中调用了 do_task()，而 do_task()可能会抛出异常，为了能捕获到 do_task()可能抛出的异常，将调用 do_task()函数的语句放在一个 **try** 关键字开头的以{}包围的 try 子句中，如果 do_task()抛出了 Some_error 异常，就会被 try 子句后的 catch 子句捕获到，catch 子句里的代码会对这个异常进行处理。即 try 子句是正常的代码，而 catch 子句是专门负责处理异常的代码。

14.2 throw、try、catch

C++的异常处理包含 3 个关键字 **throw**、**try**、**catch**。

14.2.1 throw

可以用关键字 throw 抛出任何类型的对象，这个对象称为**异常对象**。

下面的函数 g()根据输入值 i 是否 0、1 或负数而分别用 throw 抛出了不同类型的异常对象：std::string 类型的异常对象、一个自定义类 MyError 类型的异常对象、一个标准库异常类型 std::exception 的异常对象。

```
class MyError{
};
void g() {
    auto i{0};
    std::cin >> i;
    if (i == 0)
        throw std::string("I am zero");
    else if (i < 0)
        throw MyError();
    else if (i == 1)
        throw std::exception();
}
```

14.2.2 try、catch

下面的函数 f()调用了 g()函数，因为 g()可能抛出异常，为了处理 g()可能抛出的异常，需要将调用 g()的调用语句放在一个 try 关键字开头的以{}包围的 try 子句中，如：

```
void f() {
    auto j{ 3 };
    j++;
    try {
        g();    //调用的函数 g()可能会抛出异常,因此将该语句放在一个 try 子句中
        j += 2;
    }
    catch (std::string &e) {
        std::cout << e << '\n';
    }
    catch (MyError) {
        std::cout << "MyError" << '\n';
    }
    catch (...) {
        std::cout << "任何类型的异常" << '\n';
    }
}
```

try 子句中执行某条语句(如函数调用语句 g())时如果抛出了异常,则停止执行 try 子句中该语句的后续语句(如 j+=2),而用抛出的异常类型去匹配 try 子句后面的 catch 子句,即将抛出的异常对象类型和每个 catch 子句中的形参类型进行匹配,寻找最匹配的 catch 子句去处理这个异常。

假如 g()函数中输入的是 0,则该函数抛出 std::string 对象,这个对象类型就匹配 catch (std::string &e)的形参 e 的类型,即该对象被这个 catch 子句捕获到,这个 catch 子句只是简单地输出这个异常对象。如果 g()函数中输入的是负数,则抛出 MyError 类型的异常,就会被 catch (MyError)捕获到,进行相应的异常处理。如果 g()函数中输入的是 1,则抛出了 C++标准异常类型 std::exception 对象,这个对象不能被前 2 个 catch 子句捕获,但可以匹配第 3 个子句,因为这个 catch 子句的形参是 3 个点(…),表示可以匹配任何异常类型的异常。

由 try 子句和 catch 子句组成的异常处理语句也称为 **try 块**,它的定义格式如下:

```
try 子句
catch 子句序列
```

其中,**try 子句**在不同情况下可能抛出不同的异常对象;而 **catch 子句序列**则是一系列由关键字 catch 定义的 catch 子句(catch 子句也称为异常处理器)。

每个 catch 子句包含一对圆括号(),圆括号里是这个 catch 子句想捕获的异常(对象)的类型,后面是一个对这种类型异常进行处理的程序块,当然也可以是单独一条语句。

catch 圆括号内异常类型(对象)的声明类似于函数的形参参数,有 3 种不同的形式。

(1) 异常对象声明:异常对象有一个名字。catch 子句里的代码就可以从这个异常对象里得到异常的更多信息。其格式如下:

```
catch(异常对象声明) 异常处理程序块
```

例如：

```
catch (std::string &e){     std::cout << e << '\n';}
```

(2) 异常对象类型声明：只声明了异常对象的类型，没有名字。其格式如下：

```
catch(异常对象类型声明) 异常处理程序块
```

例如：

```
catch (MyError) {     std::cout << "MyError" << '\n';   }
```

(3) catch-all 声明：用 3 个小数点(…)可以捕获任何类型的异常。其格式如下：

```
catch( … )异常处理复合语句
```

例如：

```
catch (...) {std::cout << "任何类型的异常" << '\n';}
```

try 子句抛出的异常对象从上到下和每个 **catch** 子句的圆括号内的形式参数类型进行匹配，直到找到一个匹配的异常处理 catch 子句，就用这个 catch 子句处理这个异常。如果未找到匹配的异常处理 catch 子句，则该异常将传给当前函数的调用函数(外层函数)继续处理，直到最终有一个函数处理了这个异常，如果一直到最外层的 main()函数都没有处理这个异常，将调用 std::terminate()函数终止程序的执行。

因为是从上到下和每个 catch 子句中的形参匹配，因此，捕获特殊类型异常的 catch 子句在上面，捕获更一般类型异常的 catch 子句在下面，捕获所有异常的 catch(…)子句总是在最后一个。

14.2.3 异常类型的匹配

抛出的异常对象类型(假如叫 E)和 catch 子句的形参类型(假如叫 T)的匹配过程类似于函数重载解析中实参和形参类型匹配。

精确匹配：E 和 T 具有一样的类型或者 T 是 E 的左值引用类型。

基类规则：T 是 E 的无歧义基类或者 T 是 E 的无歧义基类的左值引用。

指针转换规则：T 是 U 或 const U 类型，而 U 是一个指针类型。E 是可以转换为 U 的指针类型。如 E 是一个基类指针，U 是派生类指针。

例如：

```
try {
    f();
}
catch (const std::overflow_error& e) {
    //如果 f()抛出(throw)的是 std::overflow_error 对象(相同类型)
```

```
}
catch (const std::runtime_error& e) {
    //如果 f()抛出(throw)的是 std::underflow_error (基类规则)
}
catch (const std::exception& e) {
    //如果 f()抛出(throw)的是 std::logic_error (基类规则)
}
catch (...) {
    //不管 f()抛出的是什么声明类型的异常
}
```

14.3 堆栈展开和 RAII

14.3.1 堆栈展开

C++异常机制中，当遇到 throw 语句抛出异常时，将暂停执行当前函数并搜索匹配的 catch 子句。寻找匹配的 catch 子句处理异常的过程可分为如下几种情形。

(1) 如果 throw 出现在某个 try 子句内，将检查与该 try 子句相关的一系列 catch 子句；如果有匹配的 catch 子句，则异常被该 catch 子句捕获并被处理。否则，如果没有被和该 try 块的 catch 子句捕获，则在 try 块的外层作用域去寻找能处理该异常的 catch 子句。假如这个 try 块是嵌套在另外一个 try 块中的，就在其外层 try 语句中继续搜索匹配的 catch 子句。如果一直到最外层的 try 块都没有找到匹配的 catch 子句，则退出当前函数，将控制转移到该函数的调用函数中，在调用函数中继续寻找匹配的 catch 子句；如果在调用函数中仍然没找到匹配的 catch 子句，则转移到调用函数的调用函数中寻找匹配的 catch 子句，直到最后的 main()主函数；如果直到 main()函数仍然没有找到匹配的 catch 子句，就调用 std::terminate()函数终止程序的执行。这种不断从内层 try 语句向外层 try 语句或向调用函数回退的过程类似于函数调用的栈回退过程，称为**堆栈展开**(stack unwinding)。

下面的程序中，g3()函数抛出的异常如果是 int 类型，就被 g3()函数自己的 catch 子句处理了；如果抛出的是其他异常，因为 g3()没能处理，就会将控制转移到调用 g3()的 g2()函数中，g2()函数可以处理 MyError 类型的异常。对于 g3()的其他类型异常则继续交由 g1()函数去捕获处理，g1()函数捕获了 const char * 类型的异常，但处理后，又继续抛出了 std::string 类型的异常。因为抛出异常的这个 catch 子句没有外围的 try 语句，因此，这个异常被 g1()的调用函数 main()捕获并处理。如果 g3()抛出的是 std::exception 类型的异常，在堆栈展开过程中，一直没有找到匹配的 catch 子句，因此这个异常最终没有得到处理，程序将调用 std::terminate()终止执行。

(2) 如果 throw 语句所在的函数没有 try 语句，即没有捕获异常的 catch 子句，则一旦 throw 语句抛出一个异常，就直接回退到该函数的调用函数中执行上述的**堆栈展开**过程寻找匹配的 catch 子句。如果直到最后的 main()主函数，都没有找到匹配的 catch 子句，程序调用 std::terminate()函数终止执行。

例如，14.2.2 节代码中的 g()函数的 throw 语句没有包含在一个 try 子句中，因此，程

序一旦执行 throw 语句就会回退到 g()的调用函数如 f()中寻找匹配的 catch 子句,因为 f()中调用 g()的语句位于一个 try 子句中,因此就在这个 try 子句关联的一系列 catch 子句中寻找匹配的 catch 子句。

(3) 如果没有找到一个异常处理器或者在异常被某个异常处理器捕获之前又抛出了一个异常(如进入异常处理器之前销毁的局部对象的析构函数又抛出了异常),则都会调用std::terminate()函数中止程序执行。

(4) 一个 catch 子句可能捕获了该异常,经过处理之后又抛出了这个或新的异常。那么对这个新抛出的异常也经历同样的**堆栈展开**处理过程。如果 catch 子句处理完异常但没有继续抛出异常,表示该异常得到了处理,则程序就接着这个 catch 子句所在的 try 块后面的语句继续执行。

```
#include <iostream>
#include <string>
class MyError {
};

void g3() {
    auto i{ 0 };
    std::cin >> i;
    try {
        auto d{ 0. };
        if (i == 0)
            throw "I am zero";
        else if (i < 0)
            throw MyError();
        else if (i == 1)
            throw std::exception();
        else if (i == 3)
            throw i;
        std::cout << i * i << '\n';
    }
    catch (int &e) {
        std::cout << e++ << '\t'<< e <<'\n';
    }
    std::cout << i <<'\n';
}

void g2() {
    auto j{ 3 };
    j++;
    try {
        g3();
    }
    catch(MyError e) {
        std::cout <<"MyError"<< '\n';
    }
    j *= 3;
    std::cout << j << '\n';
```

```
}

void g1() {
    //...
    try {

    }
    catch (const char * s) {
        std::cout << s << '\n';
        throw std::string(s);
    }
    //...
}
void main() {
    //...
    try {

    }
    catch (std::string s) {
        std::cout << s << '\n';
    }
    //...
}
```

异常对象对 catch 子句的形参初始化的过程和函数调用实参对形参初始化的过程是一样的。catch 子句的形参如果是引用，则直接引用抛出的异常对象。如果形参是值参数，则抛出的异常对象就会被复制给这个形式参数，因此这个异常对象的类型必须具有复制功能。

当形式参数被初始化后，堆栈展开的过程才真正开始，和 catch 匹配的 try 子句中的从开头到抛出异常之间的所有未被销毁的局部变量都会被自动销毁。

例如，g3()函数中的 catch (int &e)的 e 就是引用参数，而 catch(MyError e)的 e 就是值参数。g3()抛出异常，被一个 catch 捕获时，g3()的 try 子句的局部变量 d 就被自动销毁了。

14.3.2 资源获取即初始化

资源获取即初始化(Resource Acquisition Is Initialization，RAII)是一种 C++编程技术，通过将一个必须通过获取才能使用的资源(如动态内存、线程、网络端口、文件、互斥锁、磁盘空间、数据库连接等)绑定到一个对象的生命期，保证访问这个对象的函数都能得到资源并避免冗余的频繁检测，而当对象被销毁时，其控制的资源能得到释放，并且资源的释放次序和获取的次序正好相反(这正是析构函数和构造函数的特点)。

RAII 最初由 Bjarne Stroustrup 提出，为了保证资源的及时获取和释放，RAII 通过将资源用一个类对象封装，申请资源在类对象的构造函数中进行，而类对象被销毁时会自动调用析构函数释放资源，从而保证即使在抛出异常时，申请的资源也能得到释放。前面的智能指针就是遵循这种 RAII 的思想对动态分配内存块资源进行管理和释放。

例如，如果一个对象的构造函数在申请一个资源失败时抛出了异常，那么它的已经构造

的成员变量和基类子对象都以初始化的相反次序被销毁，从而保证这些对象申请的资源也都按相反次序正确地被释放。

具有 open()/close()、lock()/unlock()或 init()/copyFrom()/destroy()成员函数的类是典型的非 RAII 类。如标准库的互斥锁类 std::mutex 类，通过 lock()成员函数获得锁，通过 unlock()成员函数释放锁，以便保证只有获得锁的才能使用某个独占资源。

```
std::mutex m;

void bad() {
    m.lock();                        //获取这个互斥锁
    f();                             //如果 f()抛出一个异常,则这个互斥锁永远得不到释放
    if(!everything_ok()) return;     //如果这里返回,互斥锁永远得不到释放
    m.unlock();                      //只有达到这一句,互斥锁才得到释放
}
```

上述代码在获得锁(m.lock())后，进行一系列处理，但如果某个处理过程如 f()抛出了异常，将从 bad()函数中回退到其调用函数，使得互斥锁 m 占用的资源没有能得到释放，即没能执行"m.unlock();"释放其占用的资源。

解决上述问题的方法是用 RAII 技术如 std::lock_guard<std::mutex>类对互斥锁占用的资源进行管理：

```
std::mutex m;
void good(){
    std::lock_guard<std::mutex> lk(m);   //RAII 类: 互斥锁获取即初始化,即初始化对象 lk
                                         //时获取了互斥锁
    f();                                 //如果 f()抛出一个异常,互斥锁得到释放
    if(!everything_ok()) return;         //这里返回,互斥锁总得到释放
    //…
}
```

在创建 std::lock_guard<std::mutex>对象 lk 调用构造函数时自动得到了互斥锁，将来不管 f()抛出异常，还是执行"if(! everything_ok()) return;"直接返回或者函数()正常结束返回，作为局部变量的 lk 都被销毁，其析构函数自动释放其控制的互斥锁。因此，能保证对象销毁时，互斥锁总能得到释放。

下面的程序用一个互斥锁保证只有一个线程向一个文件写数据：

```
#include <mutex>
#include <iostream>
#include <string>
#include <fstream>
#include <stdexcept>

void write_to_file(const std::string & message) {
    //用于保护文件访问的互斥锁(跨线程共享)mutex
    static std::mutex mutex;
```

```
    //在访问文件之前锁定互斥锁 mutex
    std::lock_guard<std::mutex> lock(mutex);

    //试图打开文件
    std::ofstream file("example.txt");
    if (!file.is_open())
        throw std::runtime_error("unable to open file");

    file << message << std::endl;        //向文件中写信息

    //无论是否发生异常,文件将在离开范围时被关闭,
    //互斥锁 mutex 将被解锁(lock 的析构函数)
}
```

该程序采用 RAII 思想,用 std::lock_guard 对 std::mutex 互斥对象 mutex 进行包裹,无论程序是否抛出异常,作为局部变量的 std::lock_guard 在**堆栈展开**过程中都会被销毁,其析构函数都会自动解锁 mutex,从而保证该文件的控制权被释放。因此,这段代码是异常安全(exception-safe)的。

假如不使用 std::lock_guard 对象:

```
//用于保护文件访问的互斥锁(跨线程共享)mutex
static std::mutex mutex;
void write_to_file(const std::string & message) {
    mutex.lock();                    //调用互斥锁自身的 lock()成员函数锁定互斥锁对象 mutex

    //试图打开文件
    std::ofstream file("example.txt");
    if (!file.is_open())
        throw std::runtime_error("unable to open file");

    file << message << std::endl;        //向文件中写信息

    //无论是否发生异常,文件将在离开范围时被关闭,
    mutex.unlock();                      //异常发生时,无法到达这一句,因此,mutex 无法解锁
}
```

当异常发生时,因为无法到达 mutex.unlock()也就无法解锁互斥锁,导致这个锁上的文件无法被后续的代码或其他线程使用。

再看一个例子:

```
auto get_scores(){
    auto p = new double[100];
    if (!p) throw "没有足够的内存";
    //从标准输入读取一些分数放入 p 指向的动态数组中
    for (auto i = 0; i != 100; i++) {
        auto score{0.};
        std::cin >> score;
        if (score < 0) throw "输入了一个负的分数!";
        else p[i] = score;
```

```
    }
    return p;                    //for 循环中抛出异常时,到达不了该语句,即无法返回这个指针 p
}

void fun() {
    try {
        auto scores = get_scores();
        delete[] scores;                //可能不会到达这一句,引起内存泄漏
    }
    catch (const char * e) {
        std::cout << e << "\n";
    }
}

int main() {
    fun();
    //...
}
```

当 get_scores()函数的 for 循环中抛出异常时,控制直接转移到 fun()函数中,get_scores()里的后续语句如“return p;”就没有执行,因此 fun()函数中的 scores 就得不到动态内存空间指针,异常发生时,fun()函数也无法到达语句“delete[] scores;”,导致了动态分配的内存没有得到正确的释放,引起了内存泄漏。解决的方法是采用 RAII 思想,即将内存分配和释放用一个类对象包裹起来。类的构造函数负责申请动态内存,类的析构函数负责释放内存:

```
template <class T>
class WrapPtr{
    //禁止复制和赋值: 防止资源被共享
    WrapPtr(WrapPtr const &) = delete;              //禁止复制
    WrapPtr & operator = (WrapPtr const &) = delete;//禁止赋值
public:
    WrapPtr(T *p = 0) : ptr_(p) {}
    ~WrapPtr() { std::cout << "释放指针\t"; delete ptr_; }
    //下标运算符函数
    T& operator[](size_t index) noexcept { return ptr_[index]; }
    const T& operator[](size_t index) const noexcept { return ptr_[index]; }
    T * get_data()const noexcept { return ptr_; }
private:
    T * ptr_;
};
```

可以用这个类似于 std::unique_ptr<>的类模板管理独享的动态内存资源,即用这个类对象包裹原始的指针:

```
auto get_scores() {
    auto p = new double[100];
    if (!p) throw "没有足够的内存";
```

```
    WrapPtr wp(p);                  //用 WrapPtr 类对象 wp 包裹原始指针 p
    for (auto i = 0; i != 100; i++) {
        auto score{ 0. };
        std::cin >> score;
        if (score < 0) throw "输入了一个负的分数!";
        else wp[i] = score;
    }
    return wp.get_data();           //for 中抛出异常时,到达不了这一句也没关系
    //不管是否抛出异常,wp 总会自动销毁,从而释放其管理的动态内存
}

void fun() {
    try {
        auto scores = get_scores();
    }
    catch (const char * e) {
        std::cout << e << "\n";
    }
}
```

执行上述的 main()函数，当 get_scores 的 for 循环语句中抛出异常时，作为堆栈的局部变量的 wp 自动销毁，调用其析构函数释放了其构造函数管理的动态内存，避免了内存泄漏。因此，不管是否抛出异常，wp 总会自动销毁，从而释放其管理的动态内存。

执行程序，输出结果：

```
23 45 67 -2
释放指针    输入了一个负的分数!
```

14.4 习题

1. 下面的 throw 语句抛出的是什么类型的异常？如果(2)中的 throw 语句改成“throw p;”，抛出的又是什么类型的异常？

(1)
```
std::range_error r("error");
throw r;
```

(2)
```
std::exception *p = &r;
throw *p;
```

2. 程序控制在异常发生时转移到(　　)。

A. catch 子句　　　B. main()函数

C. throw 语句　　　D. 上述都不对

3. 如果在 f()的第 3 句后的位置发生异常会发生什么？如何解决这个潜在的问题？

```
void f(int * b, int * e){
    vector<int> v(b, e);
    int *p = new int[v.size()];
    ifstream in("ints");
```

```
        //假如这里出现了异常
    }
```

4. 通用 catch 处理程序由(　　)表示。

A. catch (..,)　　B. catch (—— —)

C. catch (…)　　D. catch (void x)

5. 有关 catch 处理程序的以下陈述哪些是正确的?

(1) 必须在 try 块后立即放置。

(2) 它可以有多个参数。

(3) 每个 try 块必须只有一个 catch 处理程序。

(4) try 块可以有多个 catch 处理程序。

(5) 在 try 块之后,可以将通用 catch 处理程序放在任何位置。

A. (1),(4),(5)　　B. (1),(2),(3)

C. (1),(4)　　D. (1),(2)

6. 可能导致程序异常终止的语句代码应写在(　　)子句里。

A. try　　B. catch　　C. finally　　D. 上述都不对

7. 下列程序分别输入负整数、0、正整数,结果是什么?

```
void f2() {
    double d;
    try {
        auto i{ 0};            std::cin >> i;
        if (i < 0) throw - 1;
        else       if (i == 0) throw "zero";
        else throw 3.14;
    }
    catch (float) { std::cout << "float 异常\n"; }
}
void f() {
    try { f2(); }
    catch (int) { std::cout << "int 异常\n"; }
}
int main() {
    try {
        f();
    }
    catch (double &err) {
        std::cout << "异常:" << err << "\n";
    }
}
```

8. 下列程序的输出是什么?

```
#include < iostream >
using namespace std;
int main(){
```

```
    try  {      throw 'a';}
    catch (int param)  {
        cout << "int exceptionn";
    }
    catch (...)  {
        cout << "default exceptionn";
    }
    cout << "After Exception";
}
```

9. 下列程序的输出是什么？

```
#include <iostream>
using namespace std;
class X {
public:
    X() { cout << "Constructor of X " << endl; }
    ~X() { cout << "Destructor of X " << endl; }
};
int main() {
    try {
        X  t1;      throw 10;
    }
    catch (int i) {
        cout << "捕获 " << i << endl;
    }
}
```

参 考 文 献

[1] BJARNE S. The C++ Programming Language [M]. 4th ed. New Jersey: Addison-Wesley Professional,2013.

[2] STANLEY B L,JOSEE L,BARBARA E M. C++Primer [M]. 5th ed. New Jersey: Addison-Wesley Professional,2012.

[3] 斯坦利·李普曼,约瑟·拉乔伊,芭芭拉·默. C++ Primer 中文版[M]. 5 版. 北京:电子工业出版社,2013.

[4] HORTON,I,PETER V W. Beginning C++17:From Novice to Professional[M]. Berkeley: Apress,2018.

[5] https://www.learncpp.com/.

[6] https://www.tutorialspoint.com/cplusplus/.

[7] https://a.hwdong.com/.

[8] http://www.modernescpp.com/index.php/.